Charlie’s Angels
ALL PROJECT CODE

Audio Control

```
///////////////////////////////////////////////////////////////////////////////
//
// 6.111 Introductory Digital Systems Laboratory, Spring 2006
// Team 14 "Charlie's Angels"
//          Final Project: Piano Dance Revolution
//
//          Audio Control Modules (Lucia Tian)
//
///////////////////////////////////////////////////////////////////////////////

// Below is the Labkit Code from Nathan Ickes, modified to suit our purpose.
// The instantiations and wirings follow.

///////////////////////////////////////////////////////////////////////////////
//
// Switch Debounce Module
//
///////////////////////////////////////////////////////////////////////////////

module debounce (reset, clock, noisy, clean);

   input reset, clock, noisy;
   output clean;

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

   always @(posedge clock)
     if (reset)
       begin
              count <= 0;
              new <= noisy;
              clean <= noisy;
       end
     else if (noisy != new)
       begin
              new <= noisy;
              count <= 0;
       end
     else if (count == 270000)
       clean <= new;
     else
       count <= count+1;

endmodule
```

```
///////////////////////////////////////////////////////////////////////////////
//
// 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 labkit (beep, audio_reset_b, ac97_sdata_out, ac97_sdata_in, ac97_synch,
	       ac97_bit_clock,

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

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

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

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

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

	       clock_feedback_out, clock_feedback_in,

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

	       rs232_txd, rs232_rxd, rs232_rts, rs232_cts,

	       mouse_clock, mouse_data, keyboard_clock, keyboard_data,

	       clock_27mhz, clock1, clock2,

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

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

	       switch,

	       led,

	       user1, user2, user3, user4,

	       daughtercard,

	       systemace_data, systemace_address, systemace_ce_b,
	       systemace_we_b, systemace_oe_b, systemace_irq, systemace_mpbrdy,
                analyzer1_data, analyzer1_clock,
                analyzer2_data, analyzer2_clock,
                analyzer3_data, analyzer3_clock,
                analyzer4_data, analyzer4_clock);

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

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

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

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

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

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

input  clock_feedback_in;
output clock_feedback_out;

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

output rs232_txd, rs232_rts;
input  rs232_rxd, rs232_cts;

input  mouse_clock, mouse_data, keyboard_clock, keyboard_data;

input  clock_27mhz, clock1, clock2;

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

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

```
   output [7:0] led;

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

   inout [43:0] daughtercard;

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

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

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

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

/*	Audio connections assigned to valid outputs/inputs
   assign audio_reset_b = 1'b0;
   assign ac97_synch = 1'b0;
   assign ac97_sdata_out = 1'b0;
	ac97_sdata_in is an input
*/


   // VGA Output
   assign vga_out_red = 10'h0;
   assign vga_out_green = 10'h0;
   assign vga_out_blue = 10'h0;
   assign vga_out_sync_b = 1'b1;
   assign vga_out_blank_b = 1'b1;
   assign vga_out_pixel_clock = 1'b0;
   assign vga_out_hsync = 1'b0;
   assign vga_out_vsync = 1'b0;

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

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

   // SRAMs
   assign ram0_data = 36'hZ;
   assign ram0_address = 19'h0;
   assign ram0_adv_ld = 1'b0;
   assign ram0_clk = 1'b0;
   assign ram0_cen_b = 1'b1;
   assign ram0_ce_b = 1'b1;
   assign ram0_oe_b = 1'b1;
   assign ram0_we_b = 1'b1;
   assign ram0_bwe_b = 4'hF;

// ram1 is used by songmem module.

//   assign ram1_data = 36'hZ;
//   assign ram1_address = 19'h0;
//   assign ram1_adv_ld = 1'b0;
//   assign ram1_clk = 1'b0;
//   assign ram1_cen_b = 1'b1;
//   assign ram1_ce_b = 1'b1;
//   assign ram1_oe_b = 1'b1;
//   assign ram1_we_b = 1'b1;
//   assign ram1_bwe_b = 4'hF;

   assign clock_feedback_out = 1'b0;
   // clock_feedback_in is an input

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

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

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

   // LED Displays

   assign disp_blank = 1'b1;
   assign disp_clock = 1'b0;
   assign disp_rs = 1'b0;
   assign disp_ce_b = 1'b1;
   assign disp_reset_b = 1'b0;
   assign disp_data_out = 1'b0;

   // disp_data_in is an input

   // Buttons, Switches, and Individual LEDs
   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;
*/

// Buttons used for testing purposes:

		wire button0_sync, button1_sync, button2_sync, button3_sync, button_enter_sync,
button_up_sync, button_down_sync, button_left_sync;
		wire button_up_pulse, button_down_pulse, button_left_pulse;


		wire writep, readp;
		wire sin, sout;
		wire [255:0] data_in, data_out;
		wire connected;

//Audio Control Wires
wire reset;
```

```
assign reset = button_enter_sync;

wire [35:0] direct_note;
assign direct_note = {24'b0, data_in[7:0], button3_sync, button2_sync, button1_sync, button0_sync};
wire [7:0] direct_bpm;
assign direct_bpm = {switch[6:4], 5'b0};
wire [3:0] song_number_in;
assign song_number_in = switch[3:0];


//SRAM stuff
wire [3:0] A_songaddr;
wire [11:0] A_noteaddr;
wire A_start, A_done, A_exists;
wire [35:0] A_note;
wire [7:0] A_duration;

wire [3:0] B_songaddr;
wire [11:0] B_noteaddr;
wire [35:0] B_note;
wire [7:0] B_duration;
wire B_start, B_done;
wire metronome_fast;

// debounced and pulsed buttons for testing puposes:

   debounce play_debouncee (1'b0, clock_27mhz, ~button_enter, button_enter_sync);
            debounce play_debounce0 (reset, clock_27mhz, ~button0, button0_sync);
            debounce play_debounce1(reset, clock_27mhz, ~button1, button1_sync);
            debounce play_debounce2(reset, clock_27mhz, ~button2, button2_sync);
            debounce play_debounce3 (reset, clock_27mhz, ~button3, button3_sync);
            debounce play_debounceu (reset, clock_27mhz, ~button_up, button_up_sync);
            debounce play_debounced(reset, clock_27mhz, ~button_down, button_down_sync);
            debounce play_debouncel(reset, clock_27mhz, ~button_left, button_left_sync);

            pulser pulseru(reset, clock_27mhz, button_up_sync, button_up_pulse);
            pulser pulserd(reset, clock_27mhz, button_down_sync, button_down_pulse);
            pulser pulserl(reset, clock_27mhz, button_left_sync, button_left_pulse);

// game mode
wire game_start; //pulse
assign game_start = button_left_pulse;
wire good_step;
assign led[7] = good_step;
wire [7:0] score;
assign led[6:0] = score[6:0];
wire gaming;
wire playback_startG;
wire playback_stopG;


game_mode gm1(clock_27mhz, reset,
                                                    game_start,                             // start
pulse from user control
                                                    direct_note,                    // user's note
```

```
                                                        A_note,                                    // note
from song_playback
                                                        metronome_fast,
                                                        playback_startG,          // pulse to play song
                                                        playback_stopG,                  // pulse to end
song
                                                        gaming,
                                                        good_step, score);        // output to display

//Audio Control Module

wire record_start;
assign record_start = switch[7] ? button_up_pulse : 0;
wire record_stop;
assign record_stop = switch[7] ? button_down_pulse : 0;
wire playback_start;
assign playback_start = switch[7] ?  0: button_up_pulse;
wire playback_stop;
assign playback_stop = switch[7] ?  0: button_down_pulse;

audio_control audio_control(clock_27mhz, reset,

                                                               // User Input
                                                               direct_note, direct_bpm, song_number_in,

                                                               record_start, record_stop,  playback_start,
playback_stop,

                                                               //Metronome
                                                               metronome_fast,

                                                               // AC'97 Stuff

                                                               audio_reset_b, ac97_sdata_out,
ac97_sdata_in, ac97_synch, ac97_bit_clock,

                                                               // SRAM Stuff
                                                               A_songaddr, A_noteaddr, A_start, A_done,
A_exists, A_note, A_duration,
                                                               B_songaddr, B_noteaddr, B_note,
B_duration, B_start, B_done);



// songmem reads and writes to SRAM

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


        wire [3:0] C_song = 0;
        wire [3:0] D_song = 0;
        wire [3:0] E_song = 0;
        wire [3:0] F_song = 0;
		wire [3:0] G_song = 0;

		wire [255:0] C_name, D_name, E_name, F_name, G_name;
		wire C_start = 0;
		wire C_done;
		wire [3:0] Z_song = 0;
		wire [255:0] Z_name = 0;
		wire Z_start = 0;
		wire Z_done;

   songmem sm1 (clock_27mhz, reset,
								ram1_clk, ram1_we_b, ram1_address, ram1_data,
ram1_cen_b,
								A_songaddr, A_noteaddr, A_start, A_done, A_exists, A_note,
A_duration,
								B_songaddr, B_noteaddr, B_note, B_duration, B_start,
B_done,
								C_song, D_song, E_song, F_song, G_song,
								C_name, D_name, E_name, F_name, G_name,
								C_start, C_done,
								Z_song, Z_name, Z_start, Z_done
								);

// Connection to interkit communication

   assign user4 = {29'hZ, connected, sout, 1'hZ};
		assign sin = user4[0];

		kitcom_in kin    (reset, clock_27mhz,  sin, writep, data_in, connected);
		kitcom_out kout  (reset, clock_27mhz, sout, readp, data_out);



// Analyzer assignments for testing
   assign analyzer1_data = {A_songaddr, A_noteaddr};
   assign analyzer1_clock = A_start;
   assign analyzer2_data = {B_songaddr, B_noteaddr};
   assign analyzer2_clock = B_start;

   assign analyzer3_data = {A_duration[7:0], A_note[3:0]};
   assign analyzer3_clock = A_done;
   assign analyzer4_data = {B_duration[7:0], B_note[3:0]};
   assign analyzer4_clock = B_done;


endmodule


`
////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    14:28:09 05/11/06
// Design Name:
```

```
// Module Name:    song_playback
// Project Name:
// Target Device:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
module song_playback(clock, reset,
                                playing,
                        playback_start,             // Input Pulses
                                playback_stop,
                                metronome_fast,                 // Input Metronome
                        note_out,            // Output to speakers
                        A_noteaddr, A_start, A_done, A_exists, A_note, A_duration); // For SRAM



                input clock, reset;
                output playing;                 // status signal
                input playback_start;
                input playback_stop;
                input metronome_fast;

                output [35:0] note_out;

//          to SRAM interface
                output [11:0] A_noteaddr;
                output A_start;
                input A_done, A_exists;
                input [35:0] A_note;
                input [7:0] A_duration;

// generate status signal
                reg playing;
                always @ (posedge clock) begin
                        if (reset) begin
                                playing <= 0;
                        end else if (playback_start) begin
                                playing <= 1;
                        end else if (playback_stop) begin
                                playing <= 0;
                        end
                end

                reg [11:0] A_noteaddr;
                reg A_start;
                reg [35:0] note_out;
                reg firstnote;                  //first note flag
                reg [7:0] holdcount;            //counting variable for ntoe duration
```

```
		always @ (posedge clock) begin

			A_start <= 0;		//usually 0

	// default reset values
			if (reset || !playing) begin
				A_noteaddr <= 12'b111111111111;// hack to start at 0
				note_out <= 0;
				firstnote <= 1;
				holdcount <= 0;
	//  count up to the duration at every metronome beat
			end else if (metronome_fast) begin
				holdcount <= holdcount + 1;
				if (!firstnote) note_out <= A_note;
	// move on to next address when note is done
				if (firstnote || holdcount == A_duration) begin
					firstnote <= 0;
					A_start <= 1;
					A_noteaddr <= A_noteaddr+1;
					holdcount <= 0;
				end

			end

		end
```

```
//testing
//output [31:0] beat_length;
//output [2:0] state;


/*
```

```
input clock_27mhz;
input reset;

input playback;                                         // play song if high
input [3:0] song_number_in;         // song # from game_mode
input      [7:0] bpm;                                            // beats per minute from songbank


output step;                                                      // high when playing, low when not
output [35:0] note_out;                     // 6'd note # for sound player & game_mode
output playback_done;                       // song is done playing

// SRAM Stuff
output [3:0] song_number_out;               // song # for songbank
output [11:0] note_address;                 // note # in song for songbank
output next_note;                                      // enable for songbank
input note_ready;                                      // data from mem is available
input [7:0] duration;                       // no of beats of note from songbank
input [35:0] note_in;                                  // 6'd matches to notes 0-35 from songbank




// SRAM Outputs
reg [3:0] song_number_out;
reg [11:0] note_address;
reg next_note;


// Outputs to top level
reg playback_done;
reg step;
reg [35:0] note_out;

// Internals:
reg [3:0] song_number_int;
reg [7:0] duration_int;
reg [35:0] note_int;
reg [7:0] bpm_int;



reg [31:0] beat_length;
wire [31:0] quot;
reg bpm_start;
wire bpm_stop;
bpmcalc bpm1 (clock_27mhz, reset, bpm_int, quot, bpm_start, bpm_stop);


reg [31:0] clk_count;          //counts up to beat length
reg [3:0] beat_count;                   //hold up to 15 beats long

// FSM
reg [2:0] state;
```

```
parameter IDLE = 3'd0;
parameter START = 3'd1;
parameter READ = 3'd2;
parameter CALC = 3'd3;
parameter PLAYBACK = 3'd4;




always @ (posedge clock_27mhz) begin
        next_note <= 0;
        bpm_start <= 0;

        if (reset)
                state <= IDLE;
        else begin
        case (state)
                IDLE:           begin
                                                song_number_out <= 0;
                                                note_address <= 0;
                                                step <= 0;
                                                note_out <= 0;
                                                clk_count <= 0;
                                                beat_count <= 0;
                                                playback_done <= 0;
                                                if (playback) state <= START;

                                end


                START:          begin
                                                next_note <= 1;
                                                song_number_out <= song_number_in;
                                                bpm_int <= bpm;
                                                note_address <= note_address + 1;
                                                state <= READ;
                                end

                READ:                   begin
                                                next_note <= 0;
                                                if (note_ready) begin
                                                        bpm_start <= 1;
                                                        state <= CALC;
                                                        note_int <= note_in;
                                                        duration_int <= duration;
                                                        end
                                                else state <= READ;
                                                end

                CALC:                   begin
                                                if (duration_int == 0)      begin
                                                        playback_done <= 1;
                                                        state <= IDLE;
                                                end
                                                else begin
                                                        if (bpm_stop) begin
								beat_length <= quot;
								state <= PLAYBACK;
							end
						end
						end

			PLAYBACK:	begin
						step <= 1;
						note_out <= note_int;

						if (beat_count == duration_int) begin
								step <= 0;
								state <= START;
						end
						else begin
								if (clk_count == beat_length) begin
									beat_count <= beat_count
+ 1;
									clk_count <= 0;
									end
								else clk_count <= clk_count + 1;
							 end
						end

		endcase
		end
end
*/
endmodule




///////////////////////////////////////////////////////////////////////////////
//
// 6.111 Introductory Digital Systems Laboratory, Spring 2006
// Team 14 "Charlie's Angels"
//		Final Project: Piano Dance Revolution
//
//		Audio Control Modules (Lucia Tian): Audio Control
//
///////////////////////////////////////////////////////////////////////////////

module audio_control(clock, reset,

							// User Input
							direct_note, direct_bpm, song_number_in,

							record_start, record_stop,  playback_start,
playback_stop,

							//Metronome
							metronome_fast,
                                                            // AC'97 Junk

                                                            audio_reset_b, ac97_sdata_out,
ac97_sdata_in, ac97_synch, ac97_bit_clock,

                                                            // SRAM Junk
                                                            A_songaddr, A_noteaddr, A_start, A_done,
A_exists, A_note, A_duration,
                                                            B_songaddr, B_noteaddr, B_note,
B_duration, B_start, B_done);

// clock & reset
input clock, reset;

// direct inputs from user interfaces
input [35:0] direct_note;
input [7:0] direct_bpm;
input [3:0] song_number_in;

input record_start, record_stop;
input playback_start, playback_stop;

// audiogen ac97 interface
output audio_reset_b;
output ac97_sdata_out;
input ac97_sdata_in;
output ac97_synch;
input ac97_bit_clock;

// SRAM Read
output [3:0] A_songaddr;
output [11:0] A_noteaddr;
output A_start;
input A_done;
input A_exists;
input [35:0] A_note;
input [7:0] A_duration;

//        SRAM Write
output [3:0] B_songaddr;
output [11:0] B_noteaddr;
output [35:0] B_note;
output [7:0] B_duration;
output B_start;
input B_done;

// The song address output to memory is directly tied to
// the user's song number input
assign A_songaddr = song_number_in;
assign B_songaddr = song_number_in;

// Metronome
output metronome_fast;              //Game Mode Block needs fast metronome.
wire metronome_slow, metronome_fast;       //Slow metronome only goes to internal audiogen.
metronome metro1 (.reset(reset), .clock(clock),
                                                  .bpm(direct_bpm),         //user's beats per minute
input
                                                  .metronome_slow(metronome_slow),
.metronome_fast(metronome_fast));

// audiogen signals
wire [35:0] aud_note;     //note to play
wire aud_metronome;            //one clock cycle long enable for metronome output

audiogen audiogen1(.reset(reset), .clock(clock),
                                                          .audio_reset_b(audio_reset_b),
                                                          .ac97_sdata_out(ac97_sdata_out),
.ac97_sdata_in(ac97_sdata_in),
                                                          .ac97_synch(ac97_synch),
.ac97_bit_clock(ac97_bit_clock),
                                                          .metronome(aud_metronome),
.note(aud_note));


// Record
wire recording;    // high when in record mode
wire [35:0] note_in;

record_mode recmode (reset, clock,
                                                  recording,                    // In record mode
                                                  record_start,     // On Pulse
                                                  record_stop,            // Off Pulse
                                                  metronome_fast,   // Input Metronome (at least 400
bpm)
                                                  note_in,          // Input from user
                                                  B_noteaddr, B_note, B_duration, B_start, B_done);
        // Output to write memory

assign note_in = direct_note;             // note to record is tied to user input

// Playback
wire playing;            // high when in playback mode
wire [35:0] playback_note;

song_playback playback1 (clock, reset,
                                      playing,                                   // In playback
mode
                        playback_start,       // One Pulse
                             playback_stop,                   // One Pulse
                             metronome_fast,          // Input Metronome 1/60 of Slow Metronome
                        playback_note,     // Output to speakers
                        A_noteaddr, A_start, A_done, A_exists, A_note, A_duration); // Read from memory


// If recording, play the slow metronome to speakers
// otherwise no metronome is heard.
assign aud_metronome = recording ? metronome_slow : 0;

// If playback, tie to note from playback module, else tie
// to user input.
assign aud_note = playing ? playback_note : direct_note;
```

```
endmodule


///////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    21:10:09 05/01/06
// Design Name:
// Module Name:    audiogen
// Project Name:
// Target Device:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
///////////////////////////////////////////////////////////////////////////////
module audiogen(reset, clock,
                                        audio_reset_b, ac97_sdata_out, ac97_sdata_in, ac97_synch,
ac97_bit_clock,  //AC97
                                        metronome, note);  // Inputs


//sfx
input reset, clock;
output audio_reset_b;
output ac97_sdata_out;
input ac97_sdata_in;
output ac97_synch;
input ac97_bit_clock;
input metronome;

//note_to_freq
input [35:0] note;

//sfx wires
wire audio_reset_b;
wire ac97_sdata_out, ac97_sdata_in;
wire ac97_synch, ac97_bit_clock;


//note_to_freq wires
wire [10:0] note1_N, note2_N;

//play
wire play;
assign play = (note != 0);

// SFX generates ac97 output signals from metronome and note_N inputs
   sfx sfx2(reset, clock, audio_reset_b, ac97_sdata_out,
                          ac97_sdata_in, ac97_synch, ac97_bit_clock, play, note1_N, note2_N, metronome);

// Translates input note number into two note_N counts for SFX
            note_to_freq note_to_freq1(clock, reset, note, note1_N, note2_N);

endmodule




////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    23:14:31 05/15/06
// Design Name:
// Module Name:    freq_lookup
// Project Name:
// Target Device:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
module freq_lookup(reset, clock, note_number, note_N);

input reset, clock;
input [5:0] note_number;     // 6bit representation of note
output [10:0] note_N;                        // Count increment for sine table lookup

reg [10:0] note_N;

// 6bit representation of notes
parameter REST = 6'd0;
parameter CN3 = 6'd15;
parameter CS3 = 6'd32;
parameter DN3 = 6'd16;
parameter DS3 = 6'd33;
parameter EN3 = 6'd17;
parameter FN3 = 6'd18;
parameter FS3 = 6'd34;
parameter GN3 = 6'd19;
parameter GS3 = 6'd35;
parameter AN3 = 6'd20;
parameter AS3 = 6'd36;
parameter BN3 = 6'd21;
parameter CN4 = 6'd8;
parameter CS4 = 6'd27;
parameter DN4 = 6'd9;
parameter DS4 = 6'd28;
parameter EN4 = 6'd10;
parameter FN4 = 6'd11;
parameter FS4 = 6'd29;
parameter GN4 = 6'd12;
parameter GS4 = 6'd30;
parameter AN4 = 6'd13;
parameter AS4 = 6'd31;
parameter BN4 = 6'd14;
parameter CN5 = 6'd1;
parameter CS5 = 6'd22;
parameter DN5 = 6'd2;
parameter DS5 = 6'd23;
parameter EN5 = 6'd3;
parameter FN5 = 6'd4;
parameter FS5 = 6'd24;
parameter GN5 = 6'd5;
parameter GS5 = 6'd25;
parameter AN5 = 6'd6;
parameter AS5 = 6'd26;
parameter BN5 = 6'd7;

// lookup table:

always @ (posedge clock)
        case (note_number)
          REST: note_N <= 11'd0;
          CN3: note_N <= 11'd179;
          CS3: note_N <= 11'd190;
          DN3: note_N <= 11'd201;
          DS3: note_N <= 11'd213;
          EN3: note_N <= 11'd225;
          FN3: note_N <= 11'd239;
          FS3: note_N <= 11'd253;
          GN3: note_N <= 11'd268;
          GS3: note_N <= 11'd284;
          AN3: note_N <= 11'd300;
          AS3: note_N <= 11'd318;
          BN3: note_N <= 11'd337;
          CN4: note_N <= 11'd358;
          CS4: note_N <= 11'd378;
          DN4: note_N <= 11'd401;
          DS4: note_N <= 11'd425;
          EN4: note_N <= 11'd451;
          FN4: note_N <= 11'd477;
          FS4: note_N <= 11'd505;
          GN4: note_N <= 11'd535;
          GS4: note_N <= 11'd567;
          AN4: note_N <= 11'd601;
          AS4: note_N <= 11'd636;
          BN4: note_N <= 11'd674;
          CN5: note_N <= 11'd714;
          CS5: note_N <= 11'd756;
          DN5: note_N <= 11'd801;
          DS5: note_N <= 11'd849;
          EN5: note_N <= 11'd900;
          FN5: note_N <= 11'd953;
          FS5: note_N <= 11'd1010;
```

```
            GN5: note_N <= 11'd1070;
            GS5: note_N <= 11'd1135;
            AN5: note_N <= 11'd1201;
            AS5: note_N <= 11'd1272;
    BN5: note_N <= 11'd1349;
                endcase
endmodule




////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    11:01:53 05/16/06
// Design Name:
// Module Name:    metronome
// Project Name:
// Target Device:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
module metronome(reset, clock, bpm, metronome_slow, metronome_fast);
   input reset;
   input clock;
   input [7:0] bpm;                          //beats per minute
   output metronome_slow, metronome_fast;            //slow = 1/60 freq of fast

          reg metronome_slow, metronome_fast;
          reg [31:0] beat_length;
          reg [7:0] oldbpm;
          reg [6:0] dvcount;

// BPM --> cycles per beat
wire aclr = 0;
wire sclr = 0;
wire rfd;
wire [7:0] remd;
wire div_ce;
wire [31:0] quot;
assign div_ce = 1;
wire [31:0] dividend = 32'd27000000;

//invoke IP CoreGen divider module
divi div1(dividend, oldbpm, quot, remd, clock, rfd, aclr, sclr, div_ce);

//Calculate beat length from bpm input
          always @ (posedge clock) begin
				oldbpm <= bpm;
				if (reset) begin
					dvcount <= 0;
				end else if (oldbpm != bpm) begin
					dvcount <= 0;
				end else if (dvcount < 100) begin	//wait 100 cycles for divi latency
					dvcount <= dvcount + 1;
				end else if (dvcount == 100) begin
					dvcount <= dvcount + 1;
					beat_length <= quot;
				end
			end

//generate fast metronome. count up to beat length
			reg [31:0] count;
			always @ (posedge clock) begin
				metronome_fast <= 0;
				if (reset) begin
					count <= 0;
				end else if (count >= beat_length) begin
					count <= 0;
					metronome_fast <= 1;
				end else begin
					count <= count + 1;
				end
			end

//pulse slow metronome once per 60 clock cycles
			reg [5:0] slowcount;
			always @ (posedge clock) begin
				metronome_slow <= 0;
				if (reset) begin
					slowcount <= 0;
				end else if (metronome_fast) begin
					if (slowcount >= 59) begin
						slowcount <= 0;
						metronome_slow <= 1;
					end else begin
						slowcount <= slowcount + 1;
					end
				end
			end


endmodule



////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    23:03:34 05/15/06
// Design Name:
// Module Name:    note_convert
// Project Name:
```

```
// Target Device:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
module note_convert(reset, clock, note, note1, note2);

input reset;
input clock;
input [35:0] note;                                    //36 bit note representation

output [5:0] note1, note2;                      //2 output 6 bit representations

reg [5:0] note1, note2;
reg [5:0] count;
reg [35:0] notelatch;
reg [35:0] old_notelatch;
reg activated;                                         // note has changed and the two output
                                                                                   //notes must be
updated
reg countn;

        always @ (posedge clock) begin

        // reset all regs to 0
                if (reset) begin
                        activated <= 0;
                        countn <= 0;
                        count <= 0;
                        notelatch <= 0;
                        old_notelatch <= 0;
                        note1 <= 0;
                        note2 <= 0;

        // new note is played --> latch in new note and activate system
                end else if (!activated && note != old_notelatch) begin
                        activated <= 1;
                        count <= 1;
                        countn <= 0;
                        notelatch <= note;
                        old_notelatch <= note;

        // shift and count notelatch. if 1 is encountered, store into notes 1 and 2.
                end else if (activated) begin
                        if (notelatch[0] == 1) begin
                                if (countn == 0) begin
                                        note1 <= count;
                                        countn <= 1;
                                end else begin
                                        note2 <= count;
                                        activated <= 0;
                                        countn <= 2;
                                end

        // stop process when count reaches the 36th bit.
                        end else if (count >= 36) begin
                                activated <= 0;
                                if (countn == 0) begin
                                        note1 <= 0;
                                        note2 <= 0;
                                end else if (countn == 1) begin
                                        note2 <= 0;
                                end
                        end
                        notelatch <= {1'b0, notelatch[35:1]};
                        count <= count + 1;
                end
        end

endmodule
////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    15:51:36 05/13/06
// Design Name:
// Module Name:    record_mode
// Project Name:
// Target Device:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
module record_mode(reset, clock,
                                                recording,
                                                record_start,           // On Pulse
                                                record_stop,                    // Off Pulse
                                                metronome_fast,         // Input Metronome
                                                note_in,                        // Input from user
                                                B_noteaddr, B_note, B_duration, B_start, B_done); //
B_songaddr must be set externally


input reset;
input clock;
output recording;               // status
input record_start, record_stop;
input metronome_fast;
input [35:0] note_in;
```

```
// To SRAM interface
output [11:0] B_noteaddr;
output [35:0] B_note;
output [7:0] B_duration;
output B_start;
input B_done;      // Ignored.


reg recording;

// Generate recording status signal
always @ (posedge clock) begin
        if (reset) begin
                recording <= 0;
        end else if (record_start) begin
                recording <= 1;
        end else if (record_stop) begin
                recording <= 0;
        end
end



reg [11:0] B_noteaddr;
reg [35:0] B_note;
reg [7:0] B_duration;
reg B_start;


reg [7:0] holdcount; // In beats
reg [35:0] old_note_in;                 //latched input
reg firstnote;                  //first note flag

always @ (posedge clock) begin

//write signal usually 0
        B_start <= 0;

// record end tag (all 1's) into last note address at stop
        if (record_stop) begin
                        B_noteaddr <= B_noteaddr + 1;
                        B_note <= 36'b111111111111111111111111111111111111;
                        B_duration <= 1;
                        B_start <= 1;

// reset to default values     when not recording
        end else if (reset || !recording) begin
                B_noteaddr <= 12'b111111111111; // Hack to start at 0.
                holdcount <= 0;
                firstnote <= 1;
                old_note_in <= 0;

// handles first note:
        end else if (firstnote) begin
                if (note_in != 0) begin
                        firstnote <= 0;
                        holdcount <= 0;
                        old_note_in <= note_in;
                end

// counts at every metronome beat for duration of note
        end else if (metronome_fast) begin
                old_note_in <= note_in;
                holdcount <= holdcount + 1;

        // send write signals when note_in changes and move on

                if (old_note_in != note_in) begin
                        // Issue memory write, go to next spot.
                        B_noteaddr <= B_noteaddr + 1;
                        B_note <= old_note_in;
                        B_duration <= holdcount+1;
                        B_start <= 1;
                        holdcount <= 0;
                end
        end
end


endmodule


/**

The 6.111 SFX (sound effects module), fall 2005
By Eric Fellheimer
AC'97 Sound driver by Nathan Ickes

reset - stop playing sound and go idle
audio_reset_b, ac97* - feed directly to corresponding signal in labkit
play - pulse high to start a sound effect
mode - selects with of the 4 sound effects to play (latched in on rising edge of play)

See the sound_fsm and individual effects modules(such as boing_fx) if you want to change which effects
are available

**/

module sfx (reset, clock,
                                audio_reset_b, ac97_sdata_out, ac97_sdata_in, ac97_synch,
ac97_bit_clock,
                                play, N1, N2, metronome);

// volume
   parameter VOL_PARAM = 5'd20;

// ac97 stuff
   input reset, clock;
   output audio_reset_b;
   output ac97_sdata_out;
   input ac97_sdata_in;
   output ac97_synch;
   input ac97_bit_clock;

// play notes
	input [10:0] N1, N2;
   input play;

//play metronome
   input metronome;

   wire ready;
   wire [7:0] command_address;
   wire [15:0] command_data;
   wire command_valid;

   wire [19:0] left_out_data;
   wire [19:0] right_out_data;
   wire [19:0] left_in_data, right_in_data;
   wire [4:0] volume;
   wire source;

   //hard code volume/source
   assign volume = VOL_PARAM; //a reasonable volume
   assign source = 1'b1; //microphone

   //
   // Reset controller
   //
   reg audio_reset_b;
   reg [9:0] reset_count;

   ////////////////////////////////////////////////////////////////////////////
   //
   // Reset Generation
   //
   // A shift register primitive is used to generate an active-high reset
   // signal that remains high for 16 clock cycles after configuration finishes
   // and the FPGA's internal clocks begin toggling.
   //
   ////////////////////////////////////////////////////////////////////////////

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

   always @(posedge clock) begin
     if (one_time_reset)
	begin
	   audio_reset_b <= 1'b0;
	   reset_count <= 0;
	end
     else if (reset_count == 1023)
	audio_reset_b <= 1'b1;
     else
	reset_count <= reset_count+1;
   end
```

```
            reg [19:0] left_out_data_clean, left_out_data1, left_out_data2;
            reg [19:0] right_out_data_clean, right_out_data1, right_out_data2;

// reg and delay outputs through 3 cycles
            always @ (posedge ac97_bit_clock) begin
                        left_out_data_clean <= left_out_data2;
                        left_out_data2 <= left_out_data1;
                        left_out_data1 <= left_out_data;
                        right_out_data_clean <= right_out_data2;
                        right_out_data2 <= right_out_data1;
                        right_out_data1 <= right_out_data;
                        end

//          ac97 interface
   ac97 ac97(ready, command_address, command_data, command_valid,
                left_out_data_clean, 1'b1, right_out_data_clean, 1'b1, left_in_data,
                right_in_data, ac97_sdata_out, ac97_sdata_in, ac97_synch,
                ac97_bit_clock);

//ac97 command generation
   ac97commands cmds(clock, ready, command_address, command_data,
                         command_valid, volume, source);

// sound generation w/ sine lookup tables
   sound_fsm sound_fsm(reset, clock, play, N1, N2, metronome, ready, left_out_data, right_out_data);

  endmodule

  module sound_fsm(reset, clock, play, N1, N2, metronome, ready, left_out_data, right_out_data);


   input reset, clock;
   input ready;
   input play;                                             //high when notes are not both 0
            input [10:0] N1, N2;                 // two note N-values based on 2^16 samples in sine table
            input metronome;                    //slow metronome input

//data outputs
   output [19:0] left_out_data;
   output [19:0] right_out_data;

//FSM states
   parameter S_IDLE = 2'd0;
   parameter S_START = 2'd1;
   parameter S_PLAY = 2'd2;


   reg [19:0] left_out_data, right_out_data;

//Separate FSM's for notes and metronome
   reg [1:0] note_state = S_IDLE;
            reg [1:0] met_state = S_IDLE;

   //control signals for note and metronome
   wire note_done1, note_done2, met_done;
   wire note_start, met_start;
   wire [19:0] pcm_out_note1, pcm_out_note2;
            wire [19:0] pcm_out_met;
   wire [20:0] pcm_out_note;
   reg done_cur;

//Generate next_sample signal
   reg old_ready;
            reg rdy;

   always @ (posedge clock) begin
            old_ready <= rdy;
                        rdy <= reset ? 0 : ready;
                        end
   assign next_sample = (rdy && ~old_ready);


//instantiate the sound modules
//          -two sin tables for notes
            SIN_fx  SIN_fx1(reset, clock, next_sample, pcm_out_note1, note_start, note_done1, N1);
            SIN_fx  SIN_fx2(reset, clock, next_sample, pcm_out_note2, note_start, note_done2, N2);
            slash_fx slash_fx(reset, clock, next_sample, pcm_out_met, met_start, met_done);

//Note FSM
   always @ (posedge clock)
   if(reset)
            note_state <= S_IDLE;
            else if (!play) note_state <= S_IDLE;
   else
     case (note_state)
      S_IDLE:
        if(play)
        begin
          note_state <= S_START;
        end
      S_START : note_state <= S_PLAY;
      S_PLAY: note_state <= (note_done1 || note_done2) ? S_IDLE : note_state;
      default: note_state <= S_IDLE;
     endcase

//Metronome FSM
   always @ (posedge clock)
   if(reset)
            met_state <= S_IDLE;
   else
     case (met_state)
      S_IDLE:
        if(metronome)
        begin
          met_state <= S_START;
        end
      S_START : met_state <= S_PLAY;
      S_PLAY: met_state <= met_done ? S_IDLE : met_state;
      default: met_state <= S_IDLE;
     endcase

// Generate Start Signals
```

```
   assign note_start = (note_state == S_START);
   assign met_start = (met_state == S_START);
   assign pcm_out_note = pcm_out_note1 + pcm_out_note2;

// Assign outputs to avged note values and metronome
   always @(note_state or met_state or pcm_out_note1 or pcm_out_note2 or pcm_out_met) begin
     if (note_state == S_PLAY)
            left_out_data = {1'b0, pcm_out_note[20:2]};
              else
        left_out_data = 20'h00000;

              if (met_state == S_PLAY)
                                 right_out_data = pcm_out_met;
               else
                                 right_out_data = left_out_data;
                      end

    //end always


endmodule

//AC 97 interface by Nathan Ickes
module ac97 (ready,
               command_address, command_data, command_valid,
               left_data, left_valid,
               right_data, right_valid,
               left_in_data, right_in_data,
               ac97_sdata_out, ac97_sdata_in, ac97_synch, ac97_bit_clock);

   output ready;
   input [7:0] command_address;
   input [15:0] command_data;
   input command_valid;
   input [19:0] left_data, right_data;
   input left_valid, right_valid;
   output [19:0] left_in_data, right_in_data;

   input ac97_sdata_in;
   input ac97_bit_clock;
   output ac97_sdata_out;
   output ac97_synch;

   reg ready;

   reg ac97_sdata_out;
   reg ac97_synch;

   reg [7:0] bit_count;

   reg [19:0] l_cmd_addr;
   reg [19:0] l_cmd_data;
   reg [19:0] l_left_data, l_right_data;
   reg l_cmd_v, l_left_v, l_right_v;
   reg [19:0] left_in_data, right_in_data;
```

```
initial begin
   ready <= 1'b0;
   // synthesis attribute init of ready is "0";
   ac97_sdata_out <= 1'b0;
   // synthesis attribute init of ac97_sdata_out is "0";
   ac97_synch <= 1'b0;
   // synthesis attribute init of ac97_synch is "0";

   bit_count <= 8'h00;
   // synthesis attribute init of bit_count is "0000";
   l_cmd_v <= 1'b0;
   // synthesis attribute init of l_cmd_v is "0";
   l_left_v <= 1'b0;
   // synthesis attribute init of l_left_v is "0";
   l_right_v <= 1'b0;
   // synthesis attribute init of l_right_v is "0";

   left_in_data <= 20'h00000;
   // synthesis attribute init of left_in_data is "00000";
   right_in_data <= 20'h00000;
   // synthesis attribute init of right_in_data is "00000";
end

always @(posedge ac97_bit_clock) begin
   // Generate the sync signal
   if (bit_count == 255)
	ac97_synch <= 1'b1;
   if (bit_count == 15)
	ac97_synch <= 1'b0;

   // Generate the ready signal
   if (bit_count == 128)
	ready <= 1'b1;
   if (bit_count == 2)
	ready <= 1'b0;

   // Latch user data at the end of each frame. This ensures that the
   // first frame after reset will be empty.
   if (bit_count == 255)
	begin
	   l_cmd_addr <= {command_address, 12'h000};
	   l_cmd_data <= {command_data, 4'h0};
	   l_cmd_v <= command_valid;
	   l_left_data <= left_data;
	   l_left_v <= left_valid;
	   l_right_data <= right_data;
	   l_right_v <= right_valid;
	end

   if ((bit_count >= 0) && (bit_count <= 15))
	// Slot 0: Tags
	case (bit_count[3:0])
	  4'h0: ac97_sdata_out <= 1'b1;      // Frame valid
	  4'h1: ac97_sdata_out <= l_cmd_v;   // Command address valid
	  4'h2: ac97_sdata_out <= l_cmd_v;   // Command data valid
	  4'h3: ac97_sdata_out <= l_left_v;  // Left data valid
```

```
               4'h4: ac97_sdata_out <= l_right_v; // Right data valid
               default: ac97_sdata_out <= 1'b0;
             endcase

     else if ((bit_count >= 16) && (bit_count <= 35))
             // Slot 1: Command address (8-bits, left justified)
             ac97_sdata_out <= l_cmd_v ? l_cmd_addr[35-bit_count] : 1'b0;

     else if ((bit_count >= 36) && (bit_count <= 55))
             // Slot 2: Command data (16-bits, left justified)
             ac97_sdata_out <= l_cmd_v ? l_cmd_data[55-bit_count] : 1'b0;

     else if ((bit_count >= 56) && (bit_count <= 75))
             begin
                // Slot 3: Left channel
                ac97_sdata_out <= l_left_v ? l_left_data[19] : 1'b0;
                l_left_data <= { l_left_data[18:0], l_left_data[19] };
             end
     else if ((bit_count >= 76) && (bit_count <= 95))
             // Slot 4: Right channel
                ac97_sdata_out <= l_right_v ? l_right_data[95-bit_count] : 1'b0;
     else
             ac97_sdata_out <= 1'b0;

     bit_count <= bit_count+1;

   end // always @ (posedge ac97_bit_clock)

   always @(negedge ac97_bit_clock) begin
     if ((bit_count >= 57) && (bit_count <= 76))
             // Slot 3: Left channel
             left_in_data <= { left_in_data[18:0], ac97_sdata_in };
     else if ((bit_count >= 77) && (bit_count <= 96))
             // Slot 4: Right channel
             right_in_data <= { right_in_data[18:0], ac97_sdata_in };
   end

endmodule

///////////////////////////////////////////////////////////////////////////////
//AC97 Interface code by Nathan Ickes

module ac97commands (clock, ready, command_address, command_data,
                     command_valid, volume, source);

   input clock;
   input ready;
   output [7:0] command_address;
   output [15:0] command_data;
   output command_valid;
   input [4:0] volume;
   input source;

   reg [23:0] command;
   reg command_valid;
reg old_ready;
reg done;
reg [3:0] state;

initial begin
   command <= 4'h0;
   // synthesis attribute init of command is "0";
   command_valid <= 1'b0;
   // synthesis attribute init of command_valid is "0";
   done <= 1'b0;
   // synthesis attribute init of done is "0";
   old_ready <= 1'b0;
   // synthesis attribute init of old_ready is "0";
   state <= 16'h0000;
   // synthesis attribute init of state is "0000";
end

assign command_address = command[23:16];
assign command_data = command[15:0];

wire [4:0] vol;
assign vol = 31-volume;

always @(posedge clock) begin
   if (ready && (!old_ready))
           state <= state+1;

   case (state)
           4'h0: // Read ID
             begin
                command <= 24'h80_0000;
                command_valid <= 1'b1;
             end
           4'h1: // Read ID
             command <= 24'h80_0000;
           4'h2: // Master volume
             command <= { 8'h02, 3'b000, vol, 3'b000, vol };
           4'h3: // Aux volume
             command <= { 8'h04, 3'b000, vol, 3'b000, vol };
           4'h4: // Mono volume
             command <= 24'h06_8000;
           4'h5: // PCM volume
             command <= 24'h18_0808;
           4'h6: // Record source select
             if (source)
               command <= 24'h1A_0000; // microphone
             else
               command <= 24'h1A_0404; // line-in
           4'h7: // Record gain
             command <= 24'h1C_0000;
           4'h8: // Line in gain
             command <= 24'h10_8000;
           //4'h9: // Set jack sense pins
             //command <= 24'h72_3F00;
           4'hA: // Set beep volume
             command <= 24'h0A_0000;
```

```
              //4'hF: // Misc control bits
                //command <= 24'h76_8000;
              default:
                command <= 24'h80_0000;
      endcase // case(state)

      old_ready <= ready;

   end // always @ (posedge clock)

endmodule // ac97commands


// Sine lookup module
 module SIN_fx (reset, clock, next_sample, pcm_data, start, done, N);

   input reset;
   input clock;
   input next_sample;         //ready signal from AC97
   input start;
              input [10:0] N;       //count up to this value

   output [19:0] pcm_data;
   output done;

   reg old_ready;
   reg [19:0] pcm_data;
   reg [15:0] count;
              reg disabled;

              wire [7:0] theta;    //lookup input
              wire [15:0] sine;    //sine output

//instance of IP CoreGen Sine Lookup Table
easysin easysin1(theta, clock, sine);


   always @ (posedge clock)
   begin

      if(reset) begin
                 count <= 0;
                         disabled <= 1;
                         pcm_data <= 0;
                         end

//start counting at start signal
      else if(start) begin
                 count <= 0;
                         disabled <= 0;
                         pcm_data <= 0;
                         end

// assign pcm data and increment count
      else if (next_sample) begin
                          pcm_data <= {sine, 4'b0};
```

```
                    count <= (disabled) ? count : count + N;
                                end
   end
   assign done = disabled;
              assign theta = count[15:8];

endmodule

//metronome sound effect:

module slash_fx (reset, clock, next_sample, pcm_data, start, done);

   input reset;
   input clock;
   input next_sample;
   input start;

   output [19:0] pcm_data;
   output done;

   reg old_ready;
   reg [19:0] pcm_data;
   reg [25:0] count;

   parameter seconds = 1;

// Decreased LAST_COUNT to generate shroter pulse.
   parameter LAST_COUNT = 2000 * seconds;


   always @ (posedge clock)
   begin
      if(reset)
                  count <= LAST_COUNT;
      if(start)
                  count <= 0;

      else if (next_sample)
                   count <= (done) ? count : count + 1;
   end

   assign done = (count >= LAST_COUNT);



  reg [19:0] INC = 2000;

  reg up;

  always @ (posedge clock)
  begin
    if(start)
    begin
       pcm_data <= 20'h05555;
                INC  <= 20'd2000;
       up <= 1;
    end


    if(next_sample)
    begin
      if(up)
          pcm_data <= pcm_data + INC;
                else
                   pcm_data <= pcm_data - INC;

       INC <= INC + 100;
    end

    if (up && pcm_data >=  20'hF0F00)
         up <= ~up;

    if (~up && pcm_data <= 20'h05555)
         up <= ~up;
   end
endmodule
```

Interkit Communication Code

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

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

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

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

                      tv_in_ycrcb, tv_in_data_valid, tv_in_line_clock1,
                      tv_in_line_clock2, tv_in_aef, tv_in_hff, tv_in_aff,
                      tv_in_i2c_clock, tv_in_i2c_data, tv_in_fifo_read,
                      tv_in_fifo_clock, tv_in_iso, tv_in_reset_b, tv_in_clock,
                      ram0_data, ram0_address, ram0_adv_ld, ram0_clk, ram0_cen_b,
                      ram0_ce_b, ram0_oe_b, ram0_we_b, ram0_bwe_b,

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

                      clock_feedback_out, clock_feedback_in,

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

                      rs232_txd, rs232_rxd, rs232_rts, rs232_cts,

                      mouse_clock, mouse_data, keyboard_clock, keyboard_data,

                      clock_27mhz, clock1, clock2,

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

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

                      switch,

                      led,

                      user1, user2, user3, user4,

                      daughtercard,

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

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

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

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

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

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

```
inout  tv_in_i2c_data;

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

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

input  clock_feedback_in;
output clock_feedback_out;

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

output rs232_txd, rs232_rts;
input  rs232_rxd, rs232_cts;

input  mouse_clock, mouse_data, keyboard_clock, keyboard_data;

input  clock_27mhz, clock1, clock2;

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

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

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

inout [43:0] daughtercard;

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

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

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

// Audio Input and Output
```

```
assign beep= 1'b0;
assign audio_reset_b = 1'b0;
assign ac97_synch = 1'b0;
assign ac97_sdata_out = 1'b0;

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

// Video Input
assign tv_in_i2c_clock = 1'b0;
assign tv_in_fifo_read = 1'b0;
assign tv_in_fifo_clock = 1'b0;
assign tv_in_iso = 1'b0;
assign tv_in_reset_b = 1'b0;
assign tv_in_clock = 1'b0;
assign tv_in_i2c_data = 1'bZ;

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

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

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

// LED Displays
assign disp_blank = 1'b1;
assign disp_clock = 1'b0;
assign disp_rs = 1'b0;
assign disp_ce_b = 1'b1;
assign disp_reset_b = 1'b0;
assign disp_data_out = 1'b0;

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

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

// Daughtercard Connectors
assign daughtercard = 44'hZ;

// SystemACE Microprocessor Port
assign systemace_data = 16'hZ;
assign systemace_address = 7'h0;
assign systemace_ce_b = 1'b1;
assign systemace_we_b = 1'b1;
assign systemace_oe_b = 1'b1;

// Logic Analyzer
assign analyzer1_data = 16'h0;
assign analyzer1_clock = 1'b1;
assign analyzer2_data = 16'h0;
assign analyzer2_clock = 1'b1;
assign analyzer3_data = 16'h0;
assign analyzer3_clock = 1'b1;
assign analyzer4_data = 16'h0;
assign analyzer4_clock = 1'b1;

////////////////////////////////////////////////////////////////////////////
//
// Lab 4 Components
//
////////////////////////////////////////////////////////////////////////////


// VGA Output
assign vga_out_red = 10'h0;
assign vga_out_green = 10'h0;
assign vga_out_blue = 10'h0;
assign vga_out_sync_b = 1'b1;
assign vga_out_blank_b = 1'b1;
assign vga_out_pixel_clock = 1'b0;
assign vga_out_hsync = 1'b0;
```

```
   assign vga_out_vsync = 1'b0;




   wire reset;

             debounce db1 (1'b0, clock_27mhz, ~button_enter, reset);



             // Simple test for interkit communication
             // Displays bits 6 through 0 of the input serial communcation on the LED
             // Transmits the switch on bits 7 through 0 of the output serial.
             // LED light #7 is the "connected" light.

             wire writep, readp;
             wire sin, sout;
             wire [255:0] data_in, data_out;
             wire connected;

             kitcom_in kin    (reset, clock_27mhz,  sin, writep, data_in, connected);
             kitcom_out kout  (reset, clock_27mhz, sout, readp, data_out);


             assign led = {~connected, ~data_in[6:0]};
             assign data_out = {248'b0, switch};


   // User I/Os
   assign user4 = {29'hZ, connected, sout, 1'hZ};
             assign sin = user4[0];

endmodule


module kitcom_in(reset, clock, sin, writep, data, connected);
    input reset;
    input clock;  // 27 MHz
    input sin;
              output writep;
    output [255:0] data;
              output connected;

              parameter IVAL = 2700;  // 10 KHz communication
              parameter PULSELEN = 300;   // Idle Pulse Length
              parameter SEQLEN = 32;   // Number of packets per frame.

              parameter S_IDLE = 0;      // Receiving idle LOW
              parameter S_RX = 2;     // Receiving chunk
              parameter S_END = 3;    // Receiving LOW


              // Edge detection
              // Whenever a new bit is received, issues bitclock pulse
              // This pulse may be irregularly timed.
```

```
          reg oldsin;
          reg [15:0] waitcount;
          reg bitclock;
          reg sinlatch;
          wire sindeb;
          kitcom_debounce kdeb (reset, clock, sin, sindeb);
          always @ (posedge clock) begin
                    oldsin <= sindeb;
                    if (reset) begin
                         waitcount <= 0;
                         bitclock <= 0;
                         sinlatch <= 0;
                    end else if (oldsin != sindeb) begin   // EDGE!!
                         waitcount <= 0;                // Realign
                         if (waitcount >= IVAL/2) begin
                              bitclock <= 1;
                              sinlatch <= oldsin;
                         end
                    end else if (waitcount == IVAL) begin
                         waitcount <= 0;
                         bitclock <= 1;
                         sinlatch <= oldsin;
                    end else begin
                         waitcount <= waitcount + 1;
                         bitclock <= 0;
                    end
          end

          // State Machine
          reg [2:0] state;
          reg [255:0] datalatch;
          reg [15:0] statecount;
          reg [15:0] tcount;
          reg writep;
    reg [255:0] data;             // Changes on posedge writep.
          reg connected;
          always @ (posedge clock) begin
               if (reset) begin
                    state <= S_IDLE;
                    statecount <= 0;
                    datalatch <= 0;
                    connected <= 0;
               end else if (bitclock) begin  // Happens one cycle after sinlatch is latched.
                    statecount <= statecount+1;
                    writep <= 0;
                    case (state)
                    S_IDLE:
                         begin
                              if (statecount >= PULSELEN/2 && sinlatch == 1) begin
                                   state <= S_RX;
                                   statecount <= 0;
                                   tcount <= 0;
                              end else if (sinlatch == 1) begin // May not really be start. Stay
disconnected.
                                   statecount <= 0;
                                   connected <= 0;
```

```
                        end else if (statecount >= PULSELEN*2) begin // Taking too
long. Disconnect.
                            statecount <= 0;
                            connected <= 0;
                        end
                    end
                S_RX:
                    begin
                        datalatch <= {sinlatch, datalatch[255:1]};
                        if (statecount == 255) begin
                            state <= S_END;
                            statecount <= 0;
                        end
                    end
                S_END:
                    begin
                        if (tcount == SEQLEN-1) begin // Go to resync mode.
                            state <= S_IDLE;
                            statecount <= 0;
                        end else if (statecount == 0 && sinlatch == 0) begin
                            // Good so far
                        end else if (statecount == 1 && sinlatch == 1) begin
                            // Get ready for new transmission!
                            connected <= 1;
                            tcount <= tcount+1;
                            state <= S_RX;
                            statecount <= 0;
                            // Update Frame Data
                            data <= datalatch;
                            writep <= 1;
                        end else begin              // Something went wrong.
                            connected <= 0;
                            state <= S_IDLE;
                            statecount <= 0;
                        end
                    end
                endcase
            end
        end

endmodule



// Switch Debounce Module
// use your system clock for the clock input
// to produce a synchronous, debounced output
module kitcom_debounce (reset, clock, noisy, clean);
   parameter DELAY = 20;
   input reset, clock, noisy;
   output clean;

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

   always @(posedge clock)
```

```
     if (reset)
       begin
           count <= 0;
           new <= noisy;
           clean <= noisy;
       end
     else if (noisy != new)
       begin
           new <= noisy;
           count <= 0;
       end
     else if (count == DELAY)
       clean <= new;
     else
       count <= count+1;

endmodule




module kitcom_out(reset, clock, sout, readp, data);
    input reset;
    input clock; // 27 MHz
    output sout;
    output readp;
    input [255:0] data;


        // readp goes high right after we latch in data
        // Therefore, external user should change data on posedge readp

        // communication is continuous, in this sequence:
        // 1) Idle pulse LOW (300 cycles of bit clock)
        // 2) Start bit (HIGH)
        // 3) 256 bits of transmission, followed by "0,1" (stop,start)
        // 4) Repeat step 3 SEQLEN times. (ie, 32 times)
        // 5) Go to #1.

        parameter IVAL = 2700;  // 10 KHz communication
        parameter PULSELEN = 300;   // Idle Pulse Length
        parameter SEQLEN = 32;   // Number of packets per frame.

        parameter S_IDLE = 0;      // Sending idle LOW
        parameter S_START = 1;  // Sending start    HIGH
        parameter S_TX = 2;     // Transmitting chunk
        parameter S_END = 3;    // Sending stop LOW

        // Create bit clock pulse
        reg [15:0] bitclockcount;
        reg bitclock;
        always @ (posedge clock) begin
                if (reset) begin
                        bitclockcount <= 0;
                        bitclock <= 0;
                end else if (bitclockcount == IVAL-1) begin
```

```
                        bitclockcount <= 0;
                        bitclock <= 1;
                end else begin
                        bitclockcount <= bitclockcount + 1;
                        bitclock <= 0;
                end
        end


reg [3:0] state;
reg [15:0] statecount;    // Time we've been in this state.
        reg [255:0] datalatch;
        reg [15:0] tcount;
        reg readp;
        always @ (posedge clock) begin
                if (reset) begin
                        state <= S_IDLE;
                        statecount <= 0;
                        readp <= 0;
                        datalatch <= 0;
                end else if (bitclock) begin
                        statecount <= statecount+1;
                        readp <= 0;

                        case (state)
                        S_IDLE:
                                begin
                                        if (statecount == PULSELEN-1) begin
                                                state <= S_START;
                                                statecount <= 0;
                                                tcount <= 0;
                                        end
                                end
                        S_START:
                                begin
                                        state <= S_TX;
                                        statecount <= 0;
                                        datalatch <= data;
                                        readp <= 1;
                                end
                        S_TX:
                                begin
                                        datalatch <= {1'b0,datalatch[255:1]};
                                        if (statecount == 255) begin
                                                state <= S_END;
                                                statecount <= 0;
                                        end
                                end
                        S_END:
                                begin
                                        tcount <= tcount+1;
                                        if (tcount == SEQLEN-1) begin
                                                state <= S_IDLE;
                                                statecount <= 0;
                                        end else begin
                                                state <= S_START;
```

```
                                                            statecount <= 0;
                                                  end
                                        end
                              endcase
                    end
          end

          reg sout;
          always @ (state or datalatch[0]) begin
                    if (state == S_IDLE) sout = 0;
                    else if (state == S_START) sout = 1;
                    else if (state == S_TX) sout = datalatch[0];
                    else if (state == S_END) sout = 0;
                    else sout = 0;
          end

endmodule
```

Projector Display Code

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    23:53:20 05/13/06
// Design Name:
// Module Name:    beat_sel_box
// Project Name:
// Target Device:
// Tool versions:
// Description:      draw the beat meter
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
module beat_sel_box(reset, pixel_clock, pixel_count, line_count,
                                                              beat,
                                                              beat_box_on, beat_box_overlap);

input reset, pixel_clock;
input [10:0] pixel_count;
input [9:0] line_count;
input [7:0] beat;

output beat_box_on;
```

```
reg beat_box_on;
output beat_box_overlap;
reg beat_box_overlap;

wire [5:0] bar_on_border;
wire [5:0] bar_on_inside;

// draw the beat meter
rectangle hor (reset, pixel_clock, pixel_count, line_count,
                                        11'd127, 10'd350, 11'd895, 10'd360,
                                        bar_on_border[0], bar_on_inside[0]);

rectangle v10 (reset, pixel_clock, pixel_count, line_count,
                                        11'd157, 10'd330, 11'd167, 10'd390,
                                        bar_on_border[1], bar_on_inside[1]);

rectangle v100 (reset, pixel_clock, pixel_count, line_count,
                                        11'd427, 10'd330, 11'd437, 10'd390,
                                        bar_on_border[2], bar_on_inside[2]);

rectangle v200 (reset, pixel_clock, pixel_count, line_count,
                                        11'd727, 10'd330, 11'd737, 10'd390,
                                        bar_on_border[3], bar_on_inside[3]);

rectangle v250 (reset, pixel_clock, pixel_count, line_count,
                                        11'd877, 10'd330, 11'd887, 10'd390,
                                        bar_on_border[4], bar_on_inside[4]);

//reg [10:0] custom_bar;

// this rectangle will slide across the bar based on the beat number that
// the user inputs
rectangle custom (reset, pixel_clock, pixel_count, line_count,
                                         3*beat+127, 10'd330, 3*beat+137, 10'd390,
                                         bar_on_border[5], bar_on_inside[5]);

reg [4:0] count;

always @ (posedge pixel_clock)
begin
        if (reset)
                begin
//                      custom_bar <= 11'd427;          // assuming default beat is 100
                        count <= 0;
                        beat_box_on <= 0;
                        beat_box_overlap <= 0;
                end
        else if ((bar_on_border == 6'b100010) ||        // if the two bars overlap
                                (bar_on_border == 6'b100100) ||         // the beat_box_overlap is high
                                (bar_on_border == 6'b101000) ||
                                (bar_on_border == 6'b110000) ||
                                (bar_on_border == 6'b100011) ||
                                (bar_on_border == 6'b100101) ||
                                (bar_on_border == 6'b101001) ||
                                (bar_on_border == 6'b110001) ||
                                (bar_on_inside == 6'b100010) ||
                              (bar_on_inside == 6'b100100) ||
                              (bar_on_inside == 6'b101000) ||
                              (bar_on_inside == 6'b110000) ||
                              (bar_on_inside == 6'b100011) ||
                              (bar_on_inside == 6'b100101) ||
                              (bar_on_inside == 6'b101001) ||
                              (bar_on_inside == 6'b110001))
                begin
                        beat_box_on <= 0;
                        beat_box_overlap <= 1;
                end
        else if ((bar_on_border != 6'd0) ||
                              (bar_on_inside != 6'd0))
                begin
                        beat_box_overlap <= 0;
                        beat_box_on <= 1;
                end
        else
                begin
                        beat_box_overlap <= 0;
                        beat_box_on <= 0;
                end

end


endmodule
```

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    18:35:26 05/15/06
// Design Name:
// Module Name:    beat_sel_cal
// Project Name:
// Target Device:
// Tool versions:
// Description:      This module calculates the beat per minute given the user
//                                                  input determined by the step intepretation block, and
outputs
//                                                  the beat to display_logic to be displayed onto the
screen, and
//                                            outputs the final beat to the audio block to determine the
//                                                  speed of the song playback or recording
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
module beat_sel_cal(reset, pixel_clock, pixel_count, line_count, on,
                                                            right, left, enter,
                                                            beat, final_beat);

input reset, pixel_clock;
input [10:0] pixel_count;
input [9:0] line_count;
input on, right, left, enter;

// to the beat_selection module
output [7:0] beat;
reg [7:0] beat;

// to the auidio portion of the project - determined upon enter
output [7:0] final_beat;
reg [7:0] final_beat;

reg [4:0] count;

parameter [7:0] default_beat = 100;

always @ (posedge pixel_clock)
begin
          if (reset | !on)
                    begin
                              beat <= default_beat;
                              final_beat <= 0;
                              count <= 0;
                    end
          else if (enter)
                    begin
                              final_beat <= beat;
                              count <= 0;
                    end
          // if the user keeps pressing on up, this will only cause
          // every 15 frames of screen refresh
          else if ((pixel_count == 0) && (line_count == 772))
                    begin
                              if (count <= 15)
                                        count <= count + 1;
                              else
                                        begin
                                                  count <= 0;
                                                  if (right && (beat < 8'd250))                  // beat could
only
                                                            beat <= beat + 8'd10;                          //
change by 10
                                                  else if (left && (beat > 8'd10))       // maximum beat = 250
                                                            beat <= beat - 8'd10;                          //
minimum beat = 10
                                                  else
                                                            beat <= beat;
                                        end
```

```
                end
        else
                beat <= beat;


end

endmodule






`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    23:33:58 05/13/06
// Design Name:
// Module Name:    beat_selection
// Project Name:
// Target Device:
// Tool versions:
// Description:      display the beat screen with a given beat number
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
module beat_selection(reset, pixel_clock, pixel_count, line_count, on,
                                                                        beat,
                                                                        red, green, blue);

input reset, pixel_clock;
input [10:0] pixel_count;
input [9:0] line_count;
input on;
input [7:0] beat;

output [7:0] red, green, blue;
reg [7:0] red, green, blue;

// background
wire bg_on;
vga_romdisp bg (pixel_clock, pixel_count, line_count, pixel_clock, bg_on);

// text displays
wire title_on;
defparam title.NCHAR = 13;
defparam title.NCHAR_BITS = 4;
char_string_display title (pixel_clock, pixel_count, line_count,
                                                                          title_on, "SELECT A
BEAT", 11'd408, 10'd50);

// display a 3 digit number, which indicate the beat per minute
wire beatnum_on;
defparam beat_num.NCHAR = 3;
defparam beat_num.NCHAR_BITS = 2;

wire [23:0] beat_char;
wire [3:0] remone, remten, remhundred;
wire rfdx3;
numtochar num (reset, pixel_clock, beat, remone, remten, remhundred, rfdx3, beat_char);


char_string_display beat_num (pixel_clock, pixel_count, line_count,
                                                                            beatnum_on, beat_char,
11'd456, 10'd150);

// display "/min" to after the 3 digit beat number
wire beatother_on;
defparam beat_other.NCHAR = 4;
defparam beat_other.NCHAR_BITS = 3;
char_string_display beat_other (pixel_clock, pixel_count, line_count,
                                                                                  beatother_on,
"/min", 11'd504, 10'd150);

// display buttons
wire inc_border, dec_border, enter_border, return_border;
wire inc_inside, dec_inside, enter_inside, return_inside;
wire inc_text, dec_text, enter_text, return_text;

rectangle inc_r (reset, pixel_clock, pixel_count, line_count,
                                    11'd99, 10'd450, 11'd249, 10'd650,
                                    inc_border, inc_inside);
defparam inct.NCHAR = 3;
defparam inct.NCHAR_BITS = 2;
char_string_display inct (pixel_clock, pixel_count, line_count,
                                                                     inc_text, "INC", 11'd155,
10'd540);

rectangle dec_r (reset, pixel_clock, pixel_count, line_count,
                                          11'd324, 10'd450, 11'd474, 10'd650,
                                          dec_border, dec_inside);
defparam dect.NCHAR = 4;
defparam dect.NCHAR_BITS = 3;
char_string_display dect (pixel_clock, pixel_count, line_count,
                                                                          dec_text, "DEC", 11'd370,
10'd540);

rectangle enter (reset, pixel_clock, pixel_count, line_count,
                                          11'd549, 10'd450, 11'd699, 10'd650,
                                          enter_border, enter_inside);
defparam entert.NCHAR = 5;
defparam entert.NCHAR_BITS = 3;
char_string_display entert (pixel_clock, pixel_count, line_count,
                                                                    enter_text, "ENTER",
11'd580, 10'd540);

rectangle return (reset, pixel_clock, pixel_count, line_count,
                                          11'd774, 10'd450, 11'd924, 10'd650,
                                          return_border, return_inside);
defparam returnt.NCHAR = 6;
defparam returnt.NCHAR_BITS = 3;
char_string_display returnt (pixel_clock, pixel_count, line_count,
                                                                    return_text, "RETURN",
11'd795, 10'd540);

// module that draws the beat meter
wire beat_box_on, beat_box_overlap;
beat_sel_box beat_beat (reset, pixel_clock, pixel_count, line_count,     beat,
                                                            beat_box_on, beat_box_overlap);

always @ (posedge pixel_clock)              // assigning different colors
begin                                                                      // based
on the flag signals from
        if (reset | !on)                                        // each pixel
                begin
                        red <= 8'd0;
                        green <= 8'd0;
                        blue <= 8'd0;
                end
        else if (title_on)
                begin
                        red <= 8'd0;
                        green <= 8'd0;
                        blue <= 8'd0;
                end
        else if (beatother_on | beatnum_on)
                begin
                        red <= 8'd255;
                        green <= 8'd255;
                        blue <= 8'd0;
                end
        else if (inc_text | dec_text | return_text | enter_text |
                                inc_border | dec_border | return_border |
                                enter_border)
                begin
                        red <= 8'd0;
                        green <= 8'd0;
                        blue <= 8'd0;
                end
        else if (inc_inside | dec_inside | enter_inside | return_inside)
                begin
                        if (bg_on)
                                begin
                                        red <= 8'd255;
                                        green <= 8'b10000000;
                                        blue <= 8'd255;
                                end
                        else
                                begin
```

```
                                                red <= 8'd128;
                                                green <= 8'd128;
                                                blue <= 8'd128;
                                        end
                        end
                else if (beat_box_on)
                        begin
                                red <= 8'd255;
                                green <= 8'd255;
                                blue <= 8'd255;
                        end
                else if (beat_box_overlap)                              // if the custom bar overlap the
                        begin
                // set bar (at 10, 100, 200 and 250 beats)
                                red <= 8'd255;                                          // then it turns
yellow
                                green <= 8'd255;
                                blue <= 0;
                        end
                else if (bg_on)
                        begin
                                red <= 8'd255;
                                green <= 8'd0;
                                blue <= 8'd255;
                        end
                else
                        begin
                                red <= 8'd0;
                                green <= 8'd0;
                                blue <= 8'd0;
                        end
end

endmodule
```

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    18:39:34 04/13/06
// Design Name:
// Module Name:    black_key
// Project Name:
// Target Device:
// Tool versions:
// Description:      draws a black key given the row number and the colum number of the key
//
// Dependencies:
//
// Revision:
```

```
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
module black_key(reset, pixel_clock, keyrow, keycol, pixel_count, line_count,
                                        border, inside);

input reset, pixel_clock;
input [1:0] keyrow;
input [2:0] keycol;
input [10:0] pixel_count;
input [9:0] line_count;

output border, inside;

reg [10:0] top_corner_pixel, bottom_corner_pixel;
reg [9:0] top_corner_line, bottom_corner_line;

rectangle k0 (reset, pixel_clock, pixel_count, line_count,
                                top_corner_pixel, top_corner_line,
                                bottom_corner_pixel, bottom_corner_line,
                                border, inside);

always @ (posedge pixel_clock)
begin
        if (reset)
                begin
                        top_corner_pixel <= 0;
                        top_corner_line <= 0;
                        bottom_corner_pixel <= 0;
                        bottom_corner_line <= 0;
                end
        else
                begin
                        if (keycol <= 1)
                                begin
                                        top_corner_pixel <= 120*keycol + 74;
                                        bottom_corner_pixel <= 120*keycol + 167;
                                end
                        else
                                begin
                                        top_corner_pixel <= 120*(keycol + 1) + 74;
                                        bottom_corner_pixel <= 120*(keycol + 1) + 167;
                                end
                        top_corner_line <= 256*keyrow;
                        bottom_corner_line <= 256*keyrow + 133;
                end
end

endmodule
```

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    02:30:41 05/16/06
// Design Name:
// Module Name:    control_logic
// Project Name:
// Target Device:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
module control_logic (reset, pixel_clock, pixel_count, line_count,
                                          screen_num_in, up, down, right, left, select, enter, delete,
enable,
                                                      song_title_1, song_title_2, song_title_3,
song_title_4, song_title_5,
                                                      data_ready,
                                          start_on, mode_sel_on, song_sel_on, song_ent_on,
beat_sel_on, key_board_on,
                                          title_disp, key_num, title_reg, song_code, write_ready, //
for song_enter
                                          current_pos, selected_song, latched_song_title_1,
latched_song_title_2,
                                          latched_song_title_3, latched_song_title_4,
latched_song_title_5,
                                                      song1, song2, song3, song4, song5,
read_ready, // for song_select
                                          beat, final_beat); // for beat_select

input reset, pixel_clock;
input [10:0] pixel_count;
input [9:0] line_count;

input [2:0] screen_num_in; // variable signaling which screen we are in;
input up, down, right, left, select, enter, delete;
input enable; // pulsed signal, high every second

// 5 song titles from SRAM;
input [255:0] song_title_1, song_title_2, song_title_3, song_title_4, song_title_5;
// pulsed signal, high when the song titles are stable
input data_ready;

// turn on and off a particular screen
output start_on, mode_sel_on, song_sel_on, song_ent_on, beat_sel_on, key_board_on;
reg start_on, mode_sel_on, song_sel_on, song_ent_on, beat_sel_on, key_board_on;

// for song_enter display
```

```
output [255:0] title_disp;
output [4:0] key_num;

// to SRAM, title_reg is the final title of the song that the user entered
// song_code is the number corresponding to the title;
output [255:0] title_reg;
output [3:0] song_code;
// pulsed signal, high when the title_reg is ready to be written to SRAM;
output write_ready;

// for song_select display
output [3:0] current_pos;
output [255:0] latched_song_title_1, latched_song_title_2, latched_song_title_3,
               latched_song_title_4, latched_song_title_5;

// to SRAM, code names for songs that are displayed to the screen
output [3:0] song1, song2, song3, song4, song5;
// pulsed signal, high when the addresses (song1 ... song5) are stable;
output read_ready;

// final code for selected song, output to Audio portion of the project;
output [3:0] selected_song;

// for beat_select display
output [7:0] beat;

// final beat number, output to Audio portion of the project;
output [7:0] final_beat;

// these variables are internal variables responsible for displaying the open slide
// (the one that says "welcome to piano dance revolution")
reg [4:0] time_counter;
reg start;

// instantiating modules

beat_sel_cal b_sel (reset, pixel_clock, pixel_count, line_count, beat_sel_on,
                                        right, left, enter,
                                        beat, final_beat);

song_sel_cal s_sel (reset, pixel_clock, pixel_count, line_count, song_sel_on,
                                        up, down, enter,
                                        song_title_1, song_title_2, song_title_3,
song_title_4,
                                        song_title_5, data_ready,
                                        song1, song2, song3, song4, song5, read_ready,
                                        current_pos, selected_song,
                                        latched_song_title_1, latched_song_title_2,
                                        latched_song_title_3, latched_song_title_4,
                                        latched_song_title_5);

song_enter_cal s_ent_cal (reset, pixel_clock, pixel_count, line_count, song_ent_on,
                                                    up, down, right, left, enter, delete,
select,
                                                    title_disp, key_num, title_reg,
song_code, write_ready);
```

```
always @ (posedge pixel_clock)
begin
        if (reset)
                begin
                        start_on <= 0;
                        mode_sel_on <= 0;
                        song_sel_on <= 0;
                        song_ent_on <= 0;
                        beat_sel_on <= 0;
                        key_board_on <= 0;
                        time_counter <= 0;
                        start <= 0;
                end
        else if (!start)        // keep the welcome slide on for 5 seconds
                begin                           // before going into the the mode selection screen
                        if (enable && (time_counter <= 5))
                                begin
                                        start_on <= 1;
                                        time_counter <= time_counter + 1;
                                        start <= 0;
                                end
                        else if (enable && (time_counter > 5))
                                begin
                                        time_counter <= 0;
                                        start <= 1;
                                        start_on <= 0;
                                        mode_sel_on <= 1;
                                end
                        else
                                begin
                                        start_on <= 1;
                                        start <= 0;
                                end
                end
        else if (screen_num_in == 0)                    // figuring out which screen should be
                begin                                                                   //
turned on from the screen_num_in inputs
                        mode_sel_on <= 1;                                       // from step interpretation
portion
                        start_on <= 0;
                        song_sel_on <= 0;
                        song_ent_on <= 0;
                        beat_sel_on <= 0;
                        key_board_on <= 0;
                end
        else if (screen_num_in == 1)
                begin
                        mode_sel_on <= 0;
                        start_on <= 0;
                        song_sel_on <= 1;
                        song_ent_on <= 0;
                        beat_sel_on <= 0;
                        key_board_on <= 0;
                end
        else if (screen_num_in == 2)
                begin
                        mode_sel_on <= 0;
                        start_on <= 0;
                        song_sel_on <= 0;
                        song_ent_on <= 0;
                        beat_sel_on <= 1;
                        key_board_on <= 0;
                end
        else if (screen_num_in == 3)
                begin
                        mode_sel_on <= 0;
                        start_on <= 0;
                        song_sel_on <= 0;
                        song_ent_on <= 1;
                        beat_sel_on <= 0;
                        key_board_on <= 0;
                end
        else if (screen_num_in == 4)
                begin
                        mode_sel_on <= 0;
                        start_on <= 0;
                        song_sel_on <= 0;
                        song_ent_on <= 0;
                        beat_sel_on <= 0;
                        key_board_on <= 1;
                end
        else
                begin
                        mode_sel_on <= 1;
                        start_on <= 0;
                        song_sel_on <= 0;
                        song_ent_on <= 0;
                        beat_sel_on <= 0;
                        key_board_on <= 0;
                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,char_on,cstring,cx,cy);

   parameter NCHAR = 8;  // number of 8-bit characters in cstring
   parameter NCHAR_BITS = 3; // 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 char_on;// 1 if pixel is a part of the character, 0 otherwise
   input [NCHAR*8-1:0] cstring;     // character string to display
   input [10:0] cx;
   input [9:0] cy;

   // 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 @(*)
     for (n=0 ; n<8 ; n = n+1 )                          // 8 bits per character (ASCII)
       char[n] <= cstring[column*8+n];

   // 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 cpixel = (font_byte[7 - h] ^ reverse) ? 1 : 0;
   wire dispflag = ((hcount > cx) & (vcount >= cy) & (hcount <= cx+NCHAR*16)
                              & (vcount < cy + 24));
   wire char_on = dispflag ? cpixel : 0;

endmodule
```

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

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

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

endmodule
```

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    03:18:16 05/16/06
// Design Name:
// Module Name:    display_logic
// Project Name:
// Target Device:
// Tool versions:
// Description:       Takes inputs from SRAM and control_lgic, and display screens
//                                                                  accordingly
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
module display_logic (reset, pixel_clock, pixel_count, line_count,
                                                   start_on, mode_sel_on, song_sel_on, song_ent_on,
beat_sel_on, key_board_on,
                                                   title_disp, key_num, song_code, // for song_enter,
song_code temporary
                                                   current_pos, latched_song_title_1, latched_song_title_2,
                                                   latched_song_title_3, latched_song_title_4,
latched_song_title_5, // for song_select
                                                   beat, // for beat_select
                                                   keyhit, // for key_board
                                                   vga_out_red, vga_out_green, vga_out_blue);

input reset, pixel_clock;
input [10:0] pixel_count;
input [9:0] line_count;

// flag signals keeping track of which screen should be turned on
input start_on, mode_sel_on, song_sel_on, song_ent_on, beat_sel_on, key_board_on;

// for song_enter
input [255:0] title_disp;
input [4:0] key_num;
input [3:0] song_code;

// for sog_selection
input [3:0] current_pos;
input [255:0] latched_song_title_1, latched_song_title_2, latched_song_title_3, latched_song_title_4,
                              latched_song_title_5;

// for beat
input [7:0] beat;
```

```
// for lighting up the keys, one bit representing one key, if the bit is
// 1, then the key lights up
input [35:0] keyhit;

output [7:0] vga_out_red, vga_out_green, vga_out_blue;
reg [7:0] vga_out_red, vga_out_green, vga_out_blue;

wire [7:0] R0, R1, R2, R3, R4, Rstart;
wire [7:0] G0, G1, G2, G3, G4, Gstart;
wire [7:0] B0, B1, B2, B3, B4, Bstart;

// instantiating modules
open_slide open (reset, pixel_clock, pixel_count, line_count, start_on,
                                        Rstart, Gstart, Bstart);

mode_selection mode (reset, pixel_clock, pixel_count, line_count, mode_sel_on,
                                                R0, G0, B0);

song_selection s_sel (reset, pixel_clock, pixel_count, line_count, song_sel_on,
                                                        latched_song_title_1, latched_song_title_2,
latched_song_title_3,
                                                        latched_song_title_4, latched_song_title_5,
current_pos,
                                                        R1, G1, B1);

beat_selection b_sel (reset, pixel_clock, pixel_count, line_count, beat_sel_on,
                                                        beat,
                                                        R2, G2, B2);

song_enter s_ent (reset, pixel_clock, pixel_count, line_count,   song_ent_on,
                                                title_disp, key_num, song_code,
                                                R3, G3, B3);

draw_background back_ground (reset, pixel_clock, pixel_count, line_count,

24'b111111110000000000000000, keyhit, key_board_on,
                                                                        R4, G4, B4);

always @ (posedge pixel_clock)                  // selecting the outputs from the
begin                                                                           // right
modules based on the flags
        if (reset)                                                      // (on signals)
                begin
                        vga_out_red <= 8'd0;
                        vga_out_green <= 8'd0;
                        vga_out_blue <= 8'd0;
                end
        else if (start_on)
                begin
                        vga_out_red <= Rstart;
                        vga_out_green <= Gstart;
                        vga_out_blue <= Bstart;
                end
        else if (mode_sel_on)
                begin
                        vga_out_red <= R0;
                        vga_out_green <= G0;
                        vga_out_blue <= B0;
                end
        else if (song_sel_on)
                begin
                        vga_out_red <= R1;
                        vga_out_green <= G1;
                        vga_out_blue <= B1;
                end
        else if (beat_sel_on)
                begin
                        vga_out_red <= R2;
                        vga_out_green <= G2;
                        vga_out_blue <= B2;
                end
        else if (song_ent_on)
                begin
                        vga_out_red <= R3;
                        vga_out_green <= G3;
                        vga_out_blue <= B3;
                end
        else if (key_board_on)
                begin
                        vga_out_red <= R4;
                        vga_out_green <= G4;
                        vga_out_blue <= B4;
                end
end

endmodule
```

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:            Helen Liang
//
// Create Date:    22:25:15 03/01/06
// Design Name:
// Module Name:    divider
// Project Name:
// Target Device:
// Tool versions:
// Description:     Divider is responsible for properly timing the number of
//                              seconds in every traffic light state.  It uses the 27mHz
//                                  global clock input and generate a 1Hz enable from it.
The
//                                  enable is a pulse signal that is high for only one
clock cycle
//                                  every second.
//
// Dependencies:
```

```
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//                  *note 1: since the count started at 0, it will have counted period times
//                                     when the count reaches period - 1
////////////////////////////////////////////////////////////////////////////////
module divider(clk, reset_sync, enable);

input clk;
input reset_sync;
output enable;

reg enable;
reg [25:0] count;


parameter [25:0] period = 27000000;
//parameter [3:0] period = 10;           // used for test bench

always @ (posedge clk)
begin
        if (reset_sync)
                begin
                        enable <= 0;
                        count <= 0;
                end
        else if (count < (period-1))        //*note 1
                begin
                        count <= count + 1;
                        enable <= 0;
                end
        else if (count == (period-1))
                begin
                        enable <= 1;
                        count <= 0;
                end
end

endmodule
```

```
// The synopsys directives "translate_off/translate_on" specified below are
// supported by XST, FPGA Compiler II, Mentor Graphics and Synplicity synthesis
// tools. Ensure they are correct for your synthesis tool(s).

// You must compile the wrapper file division.v when simulating
// the core, division. When compiling the wrapper file, be sure to
// reference the XilinxCoreLib Verilog simulation library. For detailed
// instructions, please refer to the "CORE Generator Help".

`timescale 1ns/1ps

module division(
	dividend,
	divisor,
	quot,
	remd,
	clk,
	rfd,
	aclr,
	sclr,
	ce);


input [7 : 0] dividend;
input [3 : 0] divisor;
output [7 : 0] quot;
output [3 : 0] remd;
input clk;
output rfd;
input aclr;
input sclr;
input ce;

// synopsys translate_off

   SDIVIDER_V3_0 #(
		1,	// c_has_aclr
```

```
                1,          // c_has_ce
                1,          // c_has_sclr
                1,          // c_sync_enable
                1,          // divclk_sel
                8,          // dividend_width
                4,          // divisor_width
                0,          // fractional_b
                4,          // fractional_width
                0)          // signed_b
        inst (
                .DIVIDEND(dividend),
                .DIVISOR(divisor),
                .QUOT(quot),
                .REMD(remd),
                .CLK(clk),
                .RFD(rfd),
                .ACLR(aclr),
                .SCLR(sclr),
                .CE(ce));


// synopsys translate_on

// FPGA Express black box declaration
// synopsys attribute fpga_dont_touch "true"
// synthesis attribute fpga_dont_touch of division is "true"

// XST black box declaration
// box_type "black_box"
// synthesis attribute box_type of division is "black_box"

endmodule






`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    17:23:40 04/13/06
// Design Name:
// Module Name:    draw_background
// Project Name:
// Target Device:
// Tool versions:
// Description:      Generate the piano keyboard with three octaves, one on
//                                                              top of another
//
// Dependencies:
//
// Revision:
```

```
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
module draw_background(reset, pixel_clock, pixel_count, line_count,
                                                                RGB_in, keyhit, on,
                                                                vga_out_red, vga_out_green,
vga_out_blue);

input reset, pixel_clock;
input [10:0] pixel_count;
input [9:0] line_count;

output [7:0] vga_out_red, vga_out_green, vga_out_blue;
reg [7:0] vga_out_red, vga_out_green, vga_out_blue;

input [23:0] RGB_in;        // select the color of the key when user steps on it

// each bit represent one key, bits 20:0 represent white keys, and bits
// 35:21 represents the black half keys.  If the bit is a 1, then the key
// is stepped on.
input [35:0] keyhit;

// 0 is the screen is not to be displayed, 1 otherwise
input on;

wire [20:0] white_border, white_inside;
wire [14:0] black_border, black_inside;
wire button_border, button_inside;
wire [3:0] button_text;

// each draws a row of key
draw_row row0 (reset, pixel_clock, 2'd0, pixel_count, line_count,
                                        white_border[6:0], white_inside[6:0],
                                        black_border[4:0], black_inside[4:0]);
draw_row row1 (reset, pixel_clock, 2'd1, pixel_count, line_count,
                                        white_border[13:7], white_inside[13:7],
                                        black_border[9:5], black_inside[9:5]);
draw_row row2 (reset, pixel_clock, 2'd2, pixel_count, line_count,
                                        white_border[20:14], white_inside[20:14],
                                        black_border[14:10], black_inside[14:10]);
return_button return (reset, pixel_clock, pixel_count, line_count,
                                                               button_border, button_inside, button_text);

always @ (posedge pixel_clock)
begin
        if (reset | !on)
                begin
                        vga_out_red <= 8'd0;
                        vga_out_green <= 8'd0;
                        vga_out_blue <= 8'd0;
                end
        else
                begin
                        if ((keyhit[35:21] &        black_inside[14:0]) != 15'd0)
                                begin
```

```
                                        vga_out_red <= RGB_in[7:0];
                                        vga_out_green <= RGB_in[15:8];
                                        vga_out_blue <= RGB_in[23:16];
                                end
                        else if ((black_border != 15'd0) || (black_inside != 15'd0))
                                begin
                                        vga_out_red <= 8'd0;
                                        vga_out_green <= 8'd0;
                                        vga_out_blue <= 8'd0;
                                end
                        else if ((keyhit[20:0] & white_inside[20:0]) != 21'd0)
                                begin
                                        vga_out_red <= RGB_in[7:0];
                                        vga_out_green <= RGB_in[15:8];
                                        vga_out_blue <= RGB_in[23:16];
                                end
                        else if (white_border != 21'd0)
                                begin
                                        vga_out_red <= 8'd0;
                                        vga_out_green <= 8'd0;
                                        vga_out_blue <= 8'd0;
                                end
                        else if (white_inside != 21'd0)
                                begin
                                        vga_out_red <= 8'b11111111;
                                        vga_out_green <= 8'b11111111;
                                        vga_out_blue <= 8'b11111111;
                                end
                        else if (button_text != 0 || button_border)
                                begin
                                        vga_out_red <= 8'd0;
                                        vga_out_green <= 8'd0;
                                        vga_out_blue <= 8'd0;
                                end
                        else if (button_inside)
                                begin
                                        vga_out_red <= 8'b11111111;
                                        vga_out_green <= 8'b11111111;
                                        vga_out_blue <= 8'b11111111;
                                end
                        else
                                begin
                                        vga_out_red <= 8'd0;
                                        vga_out_green <= 8'd0;
                                        vga_out_blue <= 8'd0;
                                end
                end
end


endmodule
```

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    15:09:17 05/12/06
// Design Name:
// Module Name:    draw_display
// Project Name:
// Target Device:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
module draw_display(reset, pixel_clock, pixel_count, line_count,
                                                       RGB_in, keyhit, screen_num,        enable, up, down,
                                                       right, left, select, enter, return, delete,
                                                       vga_out_red, vga_out_green, vga_out_blue,
                                                       song_code, song_title, write_ready, beat,
code_ready);

input reset, pixel_clock;
input [10:0] pixel_count;
input [9:0] line_count;

output [7:0] vga_out_red, vga_out_green, vga_out_blue;
reg [7:0] vga_out_red, vga_out_green, vga_out_blue;

input [23:0] RGB_in;
// each bit represent one key, bits 20:0 represent white keys, and bits
// 35:21 represents the black half keys.  If the bit is a 1, then the key
// is stepped on.
input [35:0] keyhit;

//          screen_num indicates which screen to display.
// 0 - mode selection screen
// 1 - song title selection screen
// 2 - beat selection screen
// 3 - song title input screen
// 4 - key board display
input [2:0] screen_num;

input enable;

reg [2:0] count;

input up, down, right, left, select, enter, return, delete;
output [3:0] song_code;
reg [3:0] song_code;
```

```
output [255:0] song_title;
reg [255:0] song_title;

output write_ready;
reg write_ready;
output code_ready;
reg code_ready;
output [7:0] beat;

reg start_on, mode_sel_on, song_sel_on, beat_sel_on, song_ent_on, key_board_on;

wire [7:0] R0, R1, R2, R3, R4, R_start;
wire [7:0] G0, G1, G2, G3, G4, G_start;
wire [7:0] B0, B1, B2, B3, B4, B_start;

open_slide open (reset, pixel_clock, pixel_count, line_count, start_on,
                                                    R_start, G_start, B_start);

mode_selection mode (reset, pixel_clock, pixel_count, line_count, mode_sel_on,
                                                                      R0, G0, B0);

wire [3:0] song_code_sel;
wire code_ready_temp;
song_selection song (reset, pixel_clock, pixel_count, line_count, song_sel_on,
                                                                      up, down, enter, R1, G1, B1, song_code_sel,
code_ready_temp);

beat_selection beat_r (reset, pixel_clock, pixel_count, line_count, beat_sel_on,
                                                                       right, left,
                                                                       R2, G2, B2, beat);

wire [255:0] song_title_ent;
wire [3:0] song_code_ent;
wire write_ready_temp;
song_enter enter_s (reset, pixel_clock, pixel_count, line_count,          song_ent_on,
                                                    up, down, right, left, delete, select,          enter,
                                                            R3, G3, B3, song_title_ent, song_code_ent,
write_ready_temp);

draw_background keyboard (reset, pixel_clock, pixel_count, line_count,
                                                                                 RGB_in, keyhit, key_board_on,
                                                                                 R4, G4, B4);

parameter START = 0;
parameter MODE_SEL = 1;
parameter SONG_SEL = 2;
parameter BEAT_SEL = 3;
parameter SONG_ENT = 4;
parameter KEY_BOARD = 5;
reg [3:0] state;

always @ (posedge pixel_clock)
begin
        if (reset)
                begin
```

```
            vga_out_red <= R_start;
            vga_out_green <= G_start;
            vga_out_blue <= B_start;
            count <= 0;
            start_on <= 1;
            state <= START;
            write_ready <= 0;
            song_title <= 0;
            song_code <= 0;
            code_ready <= 0;
        end
    else
        case (state)
            START:          begin
                                        if (count <= 4)
                                            begin
                                                if (enable)
                                                    count <=
count + 1;

                                                vga_out_red <=
R_start;

                                                vga_out_green
<= G_start;

                                                vga_out_blue <=
B_start;

                                                state <= START;
                                            end
                                        else
                                            begin
                                                vga_out_red <=
R0;

                                                vga_out_green
<= G0;

                                                vga_out_blue <=
B0;

                                                count <= 0;
                                                state <=
MODE_SEL;

                                                mode_sel_on <=
1;

                                                start_on <= 0;
                                            end
                                    end

            MODE_SEL:       begin
                                        if (screen_num == 0)
                                            begin
                                                vga_out_red <=
R0;

                                                vga_out_green
<= G0;

                                                vga_out_blue <=
B0;

                                                state <=
MODE_SEL;
                                            end
                                        else if (screen_num == 1)
                                            begin
                                                vga_out_red <=
R1;
                                                vga_out_green
<= G1;
                                                vga_out_blue <=
B1;
                                                state <=
SONG_SEL;
                                                song_sel_on <=
1;
                                                mode_sel_on <=
0;
                                            end
                                        else if (screen_num == 3)
                                            begin
                                                vga_out_red <=
R3;
                                                vga_out_green
<= G3;
                                                vga_out_blue <=
B3;
                                                state <=
SONG_ENT;
                                                song_ent_on <=
1;
                                                mode_sel_on <=
0;
                                            end
                                        else if (screen_num == 2)
                                            begin
                                                vga_out_red <=
R2;
                                                vga_out_green
<= R2;
                                                vga_out_blue <=
R2;
                                                state <=
BEAT_SEL;
                                                song_sel_on <=
0;
                                                beat_sel_on <= 1;
                                                code_ready <= 1;
                                                song_code <=
song_code_sel;
                                            end
                                        else if (screen_num == 4)
                                            begin
                                                vga_out_red <=
R4;
                                                vga_out_green
<= R4;
                                                vga_out_blue <=
R4;
```

```
                                                                    state <=
KEY_BOARD;
                                                                    key_board_on <=
1;
                                                                    beat_sel_on <= 0;
                                                            end
                                                    end

                        SONG_SEL:       begin
                                                        if (screen_num == 1)
                                                            begin
                                                                    vga_out_red <=
R1;
                                                                    vga_out_green
<= G1;
                                                                    vga_out_blue <=
B1;
                                                                    state <=
SONG_SEL;
                                                            end
                                                        else if (screen_num == 2)
                                                            begin
                                                                    vga_out_red <=
R2;
                                                                    vga_out_green
<= R2;
                                                                    vga_out_blue <=
R2;
                                                                    state <=
BEAT_SEL;
                                                                    song_sel_on <=
0;
                                                                    beat_sel_on <= 1;
                                                                    code_ready <= 1;
                                                                    song_code <=
song_code_sel;
                                                            end
                                                        else // if (screen_num == 0)
                                                            begin
                                                                    vga_out_red <=
R0;
                                                                    vga_out_green
<= G0;
                                                                    vga_out_blue <=
B0;
                                                                    state <=
MODE_SEL;
                                                                    song_sel_on <=
0;
                                                                    mode_sel_on <=
1;
                                                                    code_ready <= 0;
                                                            end
                                                    end

                        BEAT_SEL:       begin
```

```
                                        if (screen_num == 2)
                                            begin
                                                vga_out_red <=
R2;
                                                vga_out_green
<= R2;
                                                vga_out_blue <=
R2;
                                                state <=
BEAT_SEL;
                                                code_ready <= 0;
                                            end
                                        else if (screen_num == 4)
                                            begin
                                                vga_out_red <=
R4;
                                                vga_out_green
<= R4;
                                                vga_out_blue <=
R4;
                                                state <=
KEY_BOARD;
                                                key_board_on <=
1;
                                                beat_sel_on <= 0;
                                            end
                                        else // if (screen_num == 0)
                                            begin
                                                vga_out_red <=
R0;
                                                vga_out_green
<= G0;
                                                vga_out_blue <=
B0;
                                                state <=
MODE_SEL;
                                                song_sel_on <=
0;
                                                mode_sel_on <=
1;
                                            end
                                    end

                SONG_ENT:   begin
                                        if (screen_num == 3)
                                            begin
                                                vga_out_red <=
R3;
                                                vga_out_green
<= R3;
                                                vga_out_blue <=
R3;
                                                state <=
SONG_ENT;
                                            end
                                        else if (screen_num == 4)
                                        begin
                                                vga_out_red <=
R4;
                                                vga_out_green
<= R4;
                                                vga_out_blue <=
R4;
                                                state <=
KEY_BOARD;
                                                key_board_on <=
1;
                                                song_ent_on <=
0;
                                                write_ready <= 1;
                                                song_title <=
song_title_ent;
                                                song_code <=
song_code_ent;
                                        end
                                else // if (screen_num == 0)
                                        begin
                                                vga_out_red <=
R0;
                                                vga_out_green
<= G0;
                                                vga_out_blue <=
B0;
                                                state <=
MODE_SEL;
                                                song_ent_on <=
0;
                                                mode_sel_on <=
1;
                                                write_ready <= 0;
                                        end

                        end

            KEY_BOARD:  begin
                                if (screen_num == 4)
                                        begin
                                                vga_out_red <=
R4;
                                                vga_out_green
<= R4;
                                                vga_out_blue <=
R4;
                                                state <=
KEY_BOARD;
                                                write_ready <= 0;
                                        end
                                        else // if (screen_num ==
0)
                                        begin
                                                vga_out_red <=
R0;
```

```
                                                                                vga_out_green
<= G0;
                                                                                vga_out_blue <=
B0;
                                                                                state <=
MODE_SEL;
                                                                                key_board_on <=
0;
                                                                                mode_sel_on <=
1;
                                                                        end
                                                        end

                        default:        state <= START;

                endcase
end

endmodule
```

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    18:47:52 04/13/06
// Design Name:
// Module Name:    draw_row
// Project Name:
// Target Device:
// Tool versions:
// Description:      draw a row the piano board given the row number
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
module draw_row(reset, pixel_clock, keyrow, pixel_count, line_count,
                                        wborder, winside, bborder, binside);

input reset, pixel_clock;
input [1:0] keyrow;
input [10:0] pixel_count;
input [9:0] line_count;

output [6:0] wborder, winside;
output [4:0] bborder, binside;
```

```
// instiantes all the keys
white_key wk0 (reset, pixel_clock, keyrow, 3'd0, pixel_count, line_count,
                                        wborder[0], winside[0]);
white_key wk1 (reset, pixel_clock, keyrow, 3'd1, pixel_count, line_count,
                                        wborder[1], winside[1]);
white_key wk2 (reset, pixel_clock, keyrow, 3'd2, pixel_count, line_count,
                                        wborder[2], winside[2]);
white_key wk3 (reset, pixel_clock, keyrow, 3'd3, pixel_count, line_count,
                                        wborder[3], winside[3]);
white_key wk4 (reset, pixel_clock, keyrow, 3'd4, pixel_count, line_count,
                                        wborder[4], winside[4]);
white_key wk5 (reset, pixel_clock, keyrow, 3'd5, pixel_count, line_count,
                                        wborder[5], winside[5]);
white_key wk6 (reset, pixel_clock, keyrow, 3'd6, pixel_count, line_count,
                                        wborder[6], winside[6]);
black_key bk0 (reset, pixel_clock, keyrow, 3'd0, pixel_count, line_count,
                                                bborder[0], binside[0]);
black_key bk1 (reset, pixel_clock, keyrow, 3'd1, pixel_count, line_count,
                                                bborder[1], binside[1]);
black_key bk2 (reset, pixel_clock, keyrow, 3'd2, pixel_count, line_count,
                                                bborder[2], binside[2]);
black_key bk3 (reset, pixel_clock, keyrow, 3'd3, pixel_count, line_count,
                                                bborder[3], binside[3]);
black_key bk4 (reset, pixel_clock, keyrow, 3'd4, pixel_count, line_count,
                                                bborder[4], binside[4]);


endmodule
```

```
*    appliances, devices, or systems. Use in such applications are          *
*    expressly prohibited.                                        *
*                                                           *
*    (c) Copyright 1995-2004 Xilinx, Inc.                              *
*                                             *
*******************************************************************************/
// The synopsys directives "translate_off/translate_on" specified below are
// supported by XST, FPGA Compiler II, Mentor Graphics and Synplicity synthesis
// tools. Ensure they are correct for your synthesis tool(s).

// You must compile the wrapper file flower2_bw_320_240.v when simulating
// the core, flower2_bw_320_240. When compiling the wrapper file, be sure to
// reference the XilinxCoreLib Verilog simulation library. For detailed
// instructions, please refer to the "CORE Generator Help".

`timescale 1ns/1ps

module flower2_bw_320_240(
        addr,
        clk,
        dout);


input [17 : 0] addr;
input clk;
output [0 : 0] dout;

// synopsys translate_off

   BLKMEMSP_V6_1 #(
              18,       // c_addr_width
              "0",      // c_default_data
              196608, // c_depth
              0,        // c_enable_rlocs
              0,        // c_has_default_data
              0,        // c_has_din
              0,        // c_has_en
              0,        // c_has_limit_data_pitch
              0,        // c_has_nd
              0,        // c_has_rdy
              0,        // c_has_rfd
              0,        // c_has_sinit
              0,        // c_has_we
              18,       // c_limit_data_pitch
              "flower2_bw_320_240.mif",        // c_mem_init_file
              0,        // c_pipe_stages
              0,        // c_reg_inputs
              "0",      // c_sinit_value
              1,        // c_width
              0,        // c_write_mode
              "0",      // c_ybottom_addr
              1,        // c_yclk_is_rising
              1,        // c_yen_is_high
              "hierarchy1",     // c_yhierarchy
              0,        // c_ymake_bmm
              "16kx1",// c_yprimitive_type
```

```
                1,          // c_ysinit_is_high
                "1024",  // c_ytop_addr
                0,          // c_yuse_single_primitive
                1,          // c_ywe_is_high
                1)          // c_yydisable_warnings
        inst (
                .ADDR(addr),
                .CLK(clk),
                .DOUT(dout),
                .DIN(),
                .EN(),
                .ND(),
                .RFD(),
                .RDY(),
                .SINIT(),
                .WE());


// synopsys translate_on

// FPGA Express black box declaration
// synopsys attribute fpga_dont_touch "true"
// synthesis attribute fpga_dont_touch of flower2_bw_320_240 is "true"

// XST black box declaration
// box_type "black_box"
// synthesis attribute box_type of flower2_bw_320_240 is "black_box"

endmodule
```

```
*                                                                 *
*     Xilinx products are not intended for use in life support            *
*     appliances, devices, or systems. Use in such applications are          *
*     expressly prohibited.                                          *
*                                                                 *
*     (c) Copyright 1995-2004 Xilinx, Inc.                              *
*                                                   *
*******************************************************************************/
// The synopsys directives "translate_off/translate_on" specified below are
// supported by XST, FPGA Compiler II, Mentor Graphics and Synplicity synthesis
// tools. Ensure they are correct for your synthesis tool(s).

// You must compile the wrapper file font_rom.v when simulating
// the core, font_rom. When compiling the wrapper file, be sure to
// reference the XilinxCoreLib Verilog simulation library. For detailed
// instructions, please refer to the "CORE Generator Help".

`timescale 1ns/1ps

module font_rom(
	addr,
	clk,
	dout);


input [10 : 0] addr;
input clk;
output [10 : 0] dout;

// synopsys translate_off

   BLKMEMSP_V6_1 #(
		11,	// c_addr_width
		"0",	// c_default_data
		1536,	// c_depth
		0,	// c_enable_rlocs
		0,	// c_has_default_data
		0,	// c_has_din
		0,	// c_has_en
		0,	// c_has_limit_data_pitch
		0,	// c_has_nd
		0,	// c_has_rdy
		0,	// c_has_rfd
		0,	// c_has_sinit
		0,	// c_has_we
		18,	// c_limit_data_pitch
		"font_rom.mif",	// c_mem_init_file
		0,	// c_pipe_stages
		0,	// c_reg_inputs
		"0",	// c_sinit_value
		11,	// c_width
		0,	// c_write_mode
		"0",	// c_ybottom_addr
		1,	// c_yclk_is_rising
		1,	// c_yen_is_high
		"hierarchy1",	// c_yhierarchy
```

```
                0,          // c_ymake_bmm
                "16kx1",// c_yprimitive_type
                1,          // c_ysinit_is_high
                "1024",  // c_ytop_addr
                0,          // c_yuse_single_primitive
                1,          // c_ywe_is_high
                1)          // c_yydisable_warnings
        inst (
                .ADDR(addr),
                .CLK(clk),
                .DOUT(dout),
                .DIN(),
                .EN(),
                .ND(),
                .RFD(),
                .RDY(),
                .SINIT(),
                .WE());


// synopsys translate_on

// FPGA Express black box declaration
// synopsys attribute fpga_dont_touch "true"
// synthesis attribute fpga_dont_touch of font_rom is "true"

// XST black box declaration
// box_type "black_box"
// synthesis attribute box_type of font_rom is "black_box"

endmodule
```

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    21:37:23 05/13/06
// Design Name:
// Module Name:    get_title
// Project Name:
// Target Device:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
```

```
//
//////////////////////////////////////////////////////////////////////////////
module get_title(reset, pixel_clock, song1, song2, song3, song4, song5, read_ready,
                                        song_code, title_reg, write_ready, song_title1_m,
song_title2_m,
                                        song_title3_m, song_title4_m, song_title5_m, data_ready);

input reset, pixel_clock;
input [3:0] song1, song2, song3, song4, song5;
input read_ready;
input [3:0] song_code;
input [255:0] title_reg;
input write_ready;

output [255:0] song_title1_m, song_title2_m, song_title3_m, song_title4_m, song_title5_m;
reg [255:0] song_title1_m, song_title2_m, song_title3_m, song_title4_m, song_title5_m;

reg [255:0] song_title1_temp, song_title2_temp, song_title3_temp, song_title4_temp, song_title5_temp;
//        test_bench only
//output [39:0] song_title1_m, song_title2_m, song_title3_m, song_title4_m, song_title5_m;
//reg [39:0] song_title1_m, song_title2_m, song_title3_m, song_title4_m, song_title5_m;

output data_ready;
reg data_ready;

//output [3:0] count;
//reg [3:0] count;
//output read_now;
reg read_now;

always @ (posedge pixel_clock)
begin
        if (reset)
                begin
                        song_title1_m <= 0;
                        song_title2_m <= 0;
                        song_title3_m <= 0;
                        song_title4_m <= 0;
                        song_title5_m <= 0;
                        song_title1_temp <= 0;
                        song_title2_temp <= 0;
                        song_title3_temp <= 0;
                        song_title4_temp <= 0;
                        song_title5_temp <= 0;
                        data_ready <= 0;
                        read_now <= 0;
                end
        else if (write_ready)
                begin
                        if (song_code == 0)
                                        song_title1_temp <= title_reg;
                        else if (song_code == 1)
                                song_title2_temp <= title_reg;
                        else if (song_code == 2)
                                song_title3_temp <= title_reg;
                        else if (song_code == 3)
```

```
                                song_title4_temp <= title_reg;
                        else if (song_code == 4)
                                song_title5_temp <= title_reg;
                end
        else if (read_ready)
                begin
                        song_title1_m <= song_title1_temp;
                        song_title2_m <= song_title2_temp;
                        song_title3_m <= song_title3_temp;
                        song_title4_m <= song_title4_temp;
                        song_title5_m <= song_title5_temp;
                        data_ready <= 1;
                end
        else if (data_ready)
                data_ready <= 0;
////    else if (read_now && (count == 11))
//      else if (read_now && (count == 2001))
//              begin
//                      count <= count + 1;
//                      if (song1 == 0)
//                              begin
//                                      song_title1_m <= "song0";
//                                      song_title2_m <= "song1";
//                                      song_title3_m <= "song2";
//                                      song_title4_m <= "song3";
//                                      song_title5_m <= "song4";
//                                      data_ready <= 1;
//                              end
//                      else if (song1 == 1)
//                              begin
//                                      song_title1_m <= "song1";
//                                      song_title2_m <= "song2";
//                                      song_title3_m <= "song3";
//                                      song_title4_m <= "song4";
//                                      song_title5_m <= "song5";
//                                      data_ready <= 1;
//                              end
//                      else if (song1 == 2)
//                              begin
//                                      song_title1_m <= "song2";
//                                      song_title2_m <= "song3";
//                                      song_title3_m <= "song4";
//                                      song_title4_m <= "song5";
//                                      song_title5_m <= "song6";
//                                      data_ready <= 1;
//                              end
//                      else if (song1 == 3)
//                              begin
//                                      song_title1_m <= "song3";
//                                      song_title2_m <= "song4";
//                                      song_title3_m <= "song5";
//                                      song_title4_m <= "song6";
//                                      song_title5_m <= "song7";
//                                      data_ready <= 1;
//                              end
//                      else if (song1 == 4)
```

```
//				begin
//					song_title1_m <= "song4";
//					song_title2_m <= "song5";
//					song_title3_m <= "song6";
//					song_title4_m <= "song7";
//					song_title5_m <= "song8";
//					data_ready <= 1;
//				end
//			else if (song1 == 5)
//				begin
//					song_title1_m <= "song5";
//					song_title2_m <= "song6";
//					song_title3_m <= "song7";
//					song_title4_m <= "song8";
//					song_title5_m <= "song9";
//					data_ready <= 1;
//				end
//			else if (song1 == 6)
//				begin
//					song_title1_m <= "song6";
//					song_title2_m <= "song7";
//					song_title3_m <= "song8";
//					song_title4_m <= "song9";
//					song_title5_m <= "song10";
//					data_ready <= 1;
//				end
//			else if (song1 == 7)
//				begin
//					song_title1_m <= "song7";
//					song_title2_m <= "song8";
//					song_title3_m <= "song9";
//					song_title4_m <= "song10";
//					song_title5_m <= "song11";
//					data_ready <= 1;
//				end
//			else if (song1 == 8)
//				begin
//					song_title1_m <= "song8";
//					song_title2_m <= "song9";
//					song_title3_m <= "song10";
//					song_title4_m <= "song11";
//					song_title5_m <= "song12";
//					data_ready <= 1;
//				end
//			else if (song1 == 9)
//				begin
//					song_title1_m <= "song9";
//					song_title2_m <= "song10";
//					song_title3_m <= "song11";
//					song_title4_m <= "song12";
//					song_title5_m <= "song13";
//					data_ready <= 1;
//				end
//			else if (song1 == 10)
//				begin
//					song_title1_m <= 256'd0;
```

```
//						song_title2_m <= "song11";
//						song_title3_m <= "song12";
//						song_title4_m <= "song13";
//						song_title5_m <= "song14";
//						data_ready <= 1;
//					end
//		end
//	else
//		begin
//			data_ready <= 0;
//			count <= count + 1;
//			read_now <= 0;
//		end
end

endmodule





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

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

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

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

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

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

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

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

                  rs232_txd, rs232_rxd, rs232_rts, rs232_cts,

                  mouse_clock, mouse_data, keyboard_clock, keyboard_data,

                  clock_27mhz, clock1, clock2,

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

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

                  switch,

                  led,

                  user1, user2, user3, user4,

                  daughtercard,

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

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

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

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

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

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

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

inout  [35:0] ram1_data;
```

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

input  clock_feedback_in;
output clock_feedback_out;

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

output rs232_txd, rs232_rts;
input  rs232_rxd, rs232_cts;

input  mouse_clock, mouse_data, keyboard_clock, keyboard_data;

input  clock_27mhz, clock1, clock2;

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

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

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

inout [43:0] daughtercard;

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

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

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

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

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

// Video Input
assign tv_in_i2c_clock = 1'b0;
assign tv_in_fifo_read = 1'b0;
assign tv_in_fifo_clock = 1'b0;
assign tv_in_iso = 1'b0;
assign tv_in_reset_b = 1'b0;
assign tv_in_clock = 1'b0;
assign tv_in_i2c_data = 1'bZ;

// SRAMs
assign ram0_data = 36'hZ;
assign ram0_address = 19'h0;
assign ram0_adv_ld = 1'b0;
assign ram0_clk = 1'b0;
assign ram0_cen_b = 1'b1;
assign ram0_ce_b = 1'b1;
assign ram0_oe_b = 1'b1;
assign ram0_we_b = 1'b1;
assign ram0_bwe_b = 4'hF;

/*assign ram1_data = 36'hZ;
assign ram1_address = 19'h0;
assign ram1_adv_ld = 1'b0;
assign ram1_clk = 1'b0;
assign ram1_cen_b = 1'b1;
assign ram1_ce_b = 1'b1;
assign ram1_oe_b = 1'b1;
assign ram1_we_b = 1'b1;
assign ram1_bwe_b = 4'hF;*/

assign clock_feedback_out = 1'b0;

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

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

// LED Displays
assign disp_blank = 1'b1;
```

```
   assign disp_clock = 1'b0;
   assign disp_rs = 1'b0;
   assign disp_ce_b = 1'b1;
   assign disp_reset_b = 1'b0;
   assign disp_data_out = 1'b0;

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

   // User I/Os
   assign user1 = 32'hZ;
   assign user2 = 32'hZ;
   assign user3 = 32'hZ;
   //assign user4[31:8] = 24'hZ;
	assign user4 = 32'hZ;

   // Daughtercard Connectors
   assign daughtercard = 44'hZ;

   // SystemACE Microprocessor Port
   assign systemace_data = 16'hZ;
   assign systemace_address = 7'h0;
   assign systemace_ce_b = 1'b1;
   assign systemace_we_b = 1'b1;
   assign systemace_oe_b = 1'b1;

   // Logic Analyzer
   assign analyzer1_data = 16'h0;
   assign analyzer1_clock = 1'b1;
//   assign analyzer2_data = 16'h0;
//   assign analyzer2_clock = 1'b1;
   assign analyzer3_data = 16'h0;
   assign analyzer3_clock = 1'b1;
   assign analyzer4_data = 16'h0;
   assign analyzer4_clock = 1'b1;

   ////////////////////////////////////////////////////////////////////////////
   //
   // Lab 4 Components
   //
   ////////////////////////////////////////////////////////////////////////////

   // Generate a 64.8MHz pixel clock from clock_27mhz
   //

   wire pclk, pixel_clock;
   DCM pixel_clock_dcm (.CLKIN(clock_27mhz), .CLKFX(pclk));
   // synthesis attribute CLKFX_DIVIDE of pixel_clock_dcm is 10
   // synthesis attribute CLKFX_MULTIPLY of pixel_clock_dcm is 24
   // synthesis attribute CLK_FEEDBACK of pixel_clock_dcm is NONE
	// synthesis attribute CLKIN_PERIOD of pixel_clock_dcm is 37
   BUFG pixel_clock_buf (.I(pclk), .O(pixel_clock));


    // power-on reset generation
   wire power_on_reset;    // remain high for first 16 clocks
   SRL16 reset_sr (.D(1'b0), .CLK(pixel_clock), .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_sync,user_reset;
   debounce db1(power_on_reset, pixel_clock, ~button0, user_reset);
   assign reset_sync = user_reset | power_on_reset;





wire hblank, vblank, hsync, vsync;
wire [10:0] pixel_count;
wire [9:0] line_count;
reg vga_out_hsync, vga_out_vsync;
reg [1:0] counter;
//wire reset_sync;
wire up_sync, down_sync, right_sync, left_sync, enter_sync;
wire delete_sync, select_sync, return_sync;

assign vga_out_pixel_clock = ~pixel_clock;

assign vga_out_blank_b = (hblank && vblank);

assign vga_out_sync_b = 1'b1;

debounce up (1'b0, pixel_clock, ~button_up, up_sync);

debounce down (1'b0, pixel_clock, ~button_down, down_sync);

debounce right (1'b0, pixel_clock, ~button_right, right_sync);

debounce left (1'b0, pixel_clock, ~button_left, left_sync);

debounce enter (1'b0, pixel_clock, ~button_enter, enter_sync);

debounce select (1'b0, pixel_clock, ~button3, select_sync);

debounce return (1'b0, pixel_clock, ~button2, return_sync);

debounce delete (1'b0, pixel_clock, ~button1, delete_sync);


sync_signal_generator gen (reset_sync, pixel_clock, hblank, vblank, hsync, vsync,
                                                                        pixel_count, line_count);

wire [35:0] keyhit;
assign keyhit[0] = switch[0];
assign keyhit[1] = switch[1];
assign keyhit[2] = switch[2];
assign keyhit[3] = switch[3];
assign keyhit[4] = switch[4];
assign keyhit[35:5]= 0;
```

```
wire switch5_sync, switch6_sync, switch7_sync;
debounce s1 (1'b0, pixel_clock, switch[5], switch5_sync);
debounce s2 (1'b0, pixel_clock, switch[6], switch6_sync);
debounce s3 (1'b0, pixel_clock, switch[7], switch7_sync);

wire [2:0] screen_num_in;
assign screen_num_in[0] = switch5_sync;
assign screen_num_in[1] = switch6_sync;
assign screen_num_in[2] = switch7_sync;

wire enable;
wire [7:0] beat;
wire [3:0] song_code;
wire read_ready;
wire [3:0] song1, song2, song3, song4, song5;
wire [255:0] song_title_1, song_title_2, song_title_3, song_title_4, song_title_5;
wire data_ready;
// module imitating SRAM, testing purposes only;

wire start_on, mode_sel_on, song_sel_on, song_ent_on, beat_sel_on, key_board_on;
wire [255:0] title_disp;
wire [4:0] key_num;
wire [255:0] title_reg;
wire write_ready;
wire [3:0] current_pos, selected_song;
wire [255:0] latched_song_title_1, latched_song_title_2, latched_song_title_3,
                                        latched_song_title_4, latched_song_title_5;
wire [7:0] final_beat;

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


          wire [3:0] C_song;
          wire [3:0] D_song;
          wire [3:0] E_song;
          wire [3:0] F_song;
          wire [3:0] G_song;

          wire [255:0] C_name, D_name, E_name, F_name, G_name;
          wire C_start;
          wire C_done;

          wire [3:0] Z_song;
          wire [255:0] Z_name;
          wire Z_start;
          wire Z_done;


          // Ignore this.
          wire [3:0] A_songaddr = 0;
          wire [11:0] A_noteaddr = 0;
          wire A_start = 0;
          wire A_done;
		wire A_exists;
		wire [35:0] A_note;
		wire [7:0] A_duration;

		// Ignore This.
		wire [3:0] B_songaddr = 0;
		wire [11:0] B_noteaddr = 0;
		wire [35:0] B_note = 36'd1;
		wire [7:0] B_duration = 0;
		wire B_start = 0;
		wire B_done;

		assign C_song = song1;
		assign D_song = song2;
		assign E_song = song3;
		assign F_song = song4;
		assign G_song = song5;


		assign song_title_1 = C_name;
		assign song_title_2 = D_name;
		assign song_title_3 = E_name;
		assign song_title_4 = F_name;
		assign song_title_5 = G_name;

		assign C_start = read_ready;
		assign data_ready = C_done;

		assign Z_song = song_code;
		assign Z_name = title_reg;
		assign Z_start = write_ready;


songmem sm1 (pixel_clock, reset_sync,
						ram1_clk, ram1_we_b, ram1_address, ram1_data, ram1_cen_b,
						A_songaddr, A_noteaddr, A_start, A_done, A_exists, A_note,
A_duration,
						B_songaddr, B_noteaddr, B_note, B_duration, B_start, B_done,
						C_song, D_song, E_song, F_song, G_song,
						C_name, D_name, E_name, F_name, G_name,
						C_start, C_done,
						Z_song, Z_name, Z_start, Z_done
						);

control_logic control (reset_sync, pixel_clock, pixel_count, line_count,
								screen_num_in, up_sync, down_sync, right_sync,
left_sync,
										select_sync, enter_sync, delete_sync,
enable,
										song_title_1, song_title_2, song_title_3,
song_title_4, song_title_5,
										data_ready,
								start_on, mode_sel_on, song_sel_on, song_ent_on,
beat_sel_on, key_board_on,
								title_disp, key_num, title_reg, song_code, write_ready, //
for song_enter
```

```
                                        current_pos, selected_song, latched_song_title_1,
latched_song_title_2,
                                        latched_song_title_3, latched_song_title_4,
latched_song_title_5,
                                                song1, song2, song3, song4, song5,
read_ready, // for song_select
                                        beat, final_beat);

divider divide (clock_27mhz, reset_sync, enable);



display_logic display (reset_sync, pixel_clock, pixel_count, line_count,
                                        start_on, mode_sel_on, song_sel_on, song_ent_on,
beat_sel_on, key_board_on,
                                        title_disp, key_num, song_code, // for song_enter,
song_code temporary
                                        current_pos, latched_song_title_1, latched_song_title_2,
                                        latched_song_title_3, latched_song_title_4,
latched_song_title_5, // for song_select
                                        beat, // for beat_select
                                        keyhit, // for key_board
                                        vga_out_red, vga_out_green, vga_out_blue);

assign analyzer2_data[0] = write_ready;
assign analyzer2_data[1] = read_ready;
assign analyzer2_data[15:2] = 14'd0;
assign analyzer2_clock = 1'b1;

//get_title title1 (reset_sync, pixel_clock, song1, song2, song3, song4, song5, read_ready,
//                                          song_code, title_reg, write_ready, song_title_1,
song_title_2, song_title_3,
//                                          song_title_4, song_title_5, data_ready);


//
// testing purposes:
/*
assign user4[0] = pixel_clock;
assign user4[1] = hblank;
assign user4[2] = vblank;
assign user4[3] = hsync;
assign user4[4] = vsync;
assign user4[5] = vga_out_hsync;
assign user4[6] = vga_out_vsync;
assign user4[7] = clock_27mhz;
*/

// delaying the sync signals;
reg temp1h, temp1v, temp2h, temp2v;
//reg temp3h, temp3v;

always @ (posedge pixel_clock)
begin
        temp1h <= hsync;
        temp1v <= vsync;
```

```
		temp2h <= temp1h;
		temp2v <= temp1v;

//		temp3h <= temp2h;
//		temp3v <= temp3v;

		vga_out_hsync <= temp2h;
		vga_out_vsync <= temp2v;

//		vga_out_hsync <= temp3h;
//		vga_out_vsync <= temp3v;
end





endmodule





`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company: 
// Engineer:
//
// Create Date:    16:15:31 05/12/06
// Design Name:    
// Module Name:    mode_selection
// Project Name:   
// Target Device:  
// Tool versions:  
// Description:	draw the mode selection screen
//
// Dependencies:
// 
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
// 
////////////////////////////////////////////////////////////////////////////////
module mode_selection(reset, pixel_clock, pixel_count, line_count, on,
									red, green, blue);

input reset, pixel_clock;
input [10:0] pixel_count;
input [9:0] line_count;
input on;

output [7:0] red, green, blue;
reg [7:0] red, green, blue;
```

```
// draw the background
wire bg_on;
vga_romdisp bg (pixel_clock, pixel_count, line_count, pixel_clock, bg_on);

// draw the "play mode", "game mode" and "record mode" buttons
wire play_border, play_inside, play_text;
wire game_border, game_inside, game_text;
wire rec_border, rec_inside, rec_text;

rectangle play (reset, pixel_clock, pixel_count, line_count,
                                        11'd96, 10'd128, 11'd480, 10'd352,
                                        play_border, play_inside);

defparam play_char.NCHAR = 9;
defparam play_char.NCHAR_BITS = 4;
char_string_display play_char (pixel_clock, pixel_count, line_count,
                                                                                play_text,
"PLAY MODE", 11'd220, 10'd225);

rectangle game (reset, pixel_clock, pixel_count, line_count,
                                        11'd544, 10'd128, 11'd928, 10'd352,
                                        game_border, game_inside);

defparam game_char.NCHAR = 9;
defparam game_char.NCHAR_BITS = 4;
char_string_display game_char (pixel_clock, pixel_count, line_count,
                                                                                game_text,
"GAME MODE", 11'd670, 10'd225);

rectangle rec (reset, pixel_clock, pixel_count, line_count,
                                        11'd320, 10'd448, 11'd704, 10'd672,
                                        rec_border, rec_inside);

defparam rec_char.NCHAR = 11;
defparam rec_char.NCHAR_BITS = 4;
char_string_display rec_char (pixel_clock, pixel_count, line_count,
                                                                                rec_text,
"RECORD MODE", 11'd440, 10'd550);


always @ (posedge pixel_clock)                  // assigning the pixels the right
begin
// color outputs
        if (reset)
                begin
                        red <= 8'd0;
                        green <= 8'd0;
                        blue <= 8'd0;
                end
        else if (play_border || game_border || rec_border ||
                                play_text || game_text || rec_text)
                begin
                        red <= 8'd0;
                        green <= 8'd0;
                        blue <= 8'd0;
                end
        else if (bg_on && (game_inside || play_inside || rec_inside))
                begin
                        red <= 8'd255;
                        green <= 8'b10000000;
                        blue <= 8'd255;
                end
        else if (!bg_on && (game_inside || play_inside || rec_inside))
                begin
                        red <= 8'd96;
                        green <= 8'd96;
                        blue <= 8'd96;
                end
        else if (bg_on)
                begin
                        red <= 8'd255;
                        green <= 8'd0;
                        blue <= 8'd255;
                end
        else
                begin
                        red <= 8'd0;
                        green <= 8'd0;
                        blue <= 8'd0;
                end
end
endmodule






`timescale 1ns / 1ps

module numtochar(reset, clock, num, remone, remten, remhundred, rfdx3, numout);

input clock, reset;
input [7:0] num;

//output [31:0] char;
output [23:0] numout;
output [3:0] remone, remten, remhundred;
output rfdx3;

reg [7:0] char, counter, ones, tens, hundreds;
reg [23:0] numout;

reg [7:0] top, middle, bottom;



wire rfdx1, rfdx2, rfdx3;
wire [7:0] quotone, quotten, quothundred;
wire [3:0] remone, remten, remhundred;
```

```
division onedig(num, 4'd10, quotone, remone, clock, rfdx1, 1'd0, 1'd0, 1'd1);
division tendig(quotone, 4'd10, quotten, remten, clock, rfdx2, 1'd0, 1'd0, 1'd1);
division hundreddig(quotten, 4'd10, quothundred, remhundred, clock, rfdx3, 1'd0, 1'd0, 1'd1);


always @ (posedge clock)
begin
if(reset == 1)
		numout <= 24'b0;
else
		begin
		top <= remhundred + 48;
		middle <= remten + 48;
		bottom <= remone + 48;
		numout <= {top, middle, bottom};
		end
end






endmodule





`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    15:08:45 05/12/06
// Design Name:
// Module Name:    open_slide
// Project Name:
// Target Device:
// Tool versions:
// Description:
//
// Dependencies:    draw the welcome slide
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
module open_slide(reset, pixel_clock, pixel_count, line_count,on,
										red, green, blue);

input reset, pixel_clock;
input [10:0] pixel_count;
input [9:0] line_count;
input on;
```

```
output [7:0] red, green, blue;
reg [7:0] red, green, blue;

// draw background
wire bg_on;
vga_romdisp bg (pixel_clock, pixel_count, line_count, pixel_clock, bg_on);

// draw text "WELCOME TO PIANO DANCE REVOLUTION" in two lines
wire char_on_1;
defparam welcome.NCHAR = 10;
defparam welcome.NCHAR_BITS = 4;
char_string_display welcome (pixel_clock, pixel_count, line_count, char_on_1,
                                                                        "WELCOME TO" ,
11'd450, 10'd300);
wire char_on_2;
defparam title.NCHAR = 22;
defparam title.NCHAR_BITS = 5;
char_string_display title (pixel_clock, pixel_count, line_count, char_on_2,
                                                                       "PIANO DANCE
REVOLUTION", 11'd375, 10'd400);

always @ (posedge pixel_clock)          // assigning color outputs to each pixel
begin
        if (reset | !on)
                begin
                        red <= 8'd0;
                        green <= 8'd0;
                        blue <= 8'd0;
                end
        else if (char_on_1 | char_on_2)
                begin
                        red <= 8'b11111111;
                        green <= 8'b11111111;
                        blue <= 8'b11111111;
                end
        else if (bg_on)
                begin
                        red <= 8'b11111111;
                        green <= 8'd0;
                        blue <= 8'b11111111;
                end
        else
                begin
                        red <= 8'd0;
                        green <= 8'd0;
                        blue <= 8'd0;
                end
end


endmodule
```

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    17:04:48 04/13/06
// Design Name:
// Module Name:    rectangle
// Project Name:
// Target Device:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
module rectangle(reset, pixel_clock, pixel_count, line_count,
                                        top_corner_pixel, top_corner_line,
                                        bottom_corner_pixel, bottom_corner_line,
                                        on_border, in_rectangle);

input reset, pixel_clock;

input [10:0] pixel_count;
input [9:0] line_count;
input [10:0] top_corner_pixel;
input [9:0] top_corner_line;
input [10:0] bottom_corner_pixel;
input [9:0] bottom_corner_line;

output on_border, in_rectangle;
reg on_border, in_rectangle;

always @ (posedge pixel_clock)
begin
        if (reset ||
                 (pixel_count < top_corner_pixel) ||
                 (pixel_count > bottom_corner_pixel) ||
                 (line_count < top_corner_line) ||
                 (line_count > bottom_corner_line))
                begin
                        on_border <= 0;
                        in_rectangle <= 0;
                end
        else if ((pixel_count > top_corner_pixel) &&
                                (pixel_count < bottom_corner_pixel) &&
                                (line_count > top_corner_line) &&
                                (line_count < bottom_corner_line))
                begin
                        on_border <= 0;
                        in_rectangle <= 1;
                end
        else
                begin
                        on_border <= 1;
                        in_rectangle <= 0;
                end
end


endmodule






`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    23:57:32 05/11/06
// Design Name:
// Module Name:    return_button
// Project Name:
// Target Device:
// Tool versions:
// Description:      generate the return button for the keyboard display
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
module return_button(reset, pixel_clock, pixel_count, line_count,
                                                        rb_border, rb_inside, rb_letter);

input reset, pixel_clock;
input [10:0] pixel_count;
input [9:0]         line_count;

output rb_border, rb_inside;
output [3:0] rb_letter;

// draw box for button
wire inside, letter1, letter2, letter3, letter4;
rectangle rec (reset, pixel_clock, pixel_count, line_count,
                                        11'd901, 10'd0, 11'd1021, 10'd767, rb_border, rb_inside);

// display text on top of the button
defparam line1.NCHAR = 6;
defparam line1.NCHAR_BITS = 3;
char_string_display line1 (pixel_clock, pixel_count,line_count,
                                                                                        rb_letter[0], "RETURN" ,
11'd915, 10'd300);
defparam line2.NCHAR = 2;
defparam line2.NCHAR_BITS = 2;
char_string_display line2 (pixel_clock, pixel_count,line_count,
                                                                                        rb_letter[1], "TO" ,

11'd940, 10'd325);
defparam line3.NCHAR = 4;
defparam line3.NCHAR_BITS = 3;
char_string_display line3 (pixel_clock, pixel_count,line_count,
                                                                                        rb_letter[2], "MAIN" ,

11'd930, 10'd350);
defparam line4.NCHAR = 4;
defparam line4.NCHAR_BITS = 3;
char_string_display line4 (pixel_clock, pixel_count,line_count,
                                                                                        rb_letter[3], "MENU" ,

11'd930, 10'd375);




endmodule






`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    15:36:58 05/14/06
// Design Name:
// Module Name:    song_enter
// Project Name:
// Target Device:
// Tool versions:
// Description:
//
// Dependencies:   draw the enter song title screen
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
module song_enter(reset, pixel_clock, pixel_count, line_count,        on,
                                                              title, letter, song_code,
                                                              red, green, blue);

input reset, pixel_clock;
input [10:0] pixel_count;
input [9:0] line_count;
input on;
input [255:0] title;
// 0 - 26 coding for 26 letters and space
input [4:0] letter;
input [3:0] song_code;

output [7:0] red, green, blue;
reg [7:0] red, green, blue;

// getting background from ROM;
wire bg_on;
vga_romdisp bg (pixel_clock, pixel_count, line_count, pixel_clock, bg_on);

// writing title of the slide;
wire title_on;
defparam title_c.NCHAR = 15;
defparam title_c.NCHAR_BITS = 4;
char_string_display title_c (pixel_clock, pixel_count, line_count,
                                                                        title_on, "ENTER SONG
NAME", 11'd392, 10'd50);


// generating up, down, right and left buttons;
wire up_border, down_border, right_border, left_border;
wire up_inside, down_inside, right_inside, left_inside;
wire up_text, down_text, right_text, left_text;

rectangle up_r (reset, pixel_clock, pixel_count, line_count,
                                  11'd99, 10'd592, 11'd249, 10'd752,
                                  up_border, up_inside);
defparam upt.NCHAR = 2;
defparam upt.NCHAR_BITS = 2;
char_string_display upt (pixel_clock, pixel_count, line_count,
                                                                 up_text, "UP", 11'd175, 10'd660);

rectangle down_r (reset, pixel_clock, pixel_count, line_count,
                                         11'd324, 10'd592, 11'd474, 10'd752,
                                         down_border, down_inside);
defparam downt.NCHAR = 4;
defparam downt.NCHAR_BITS = 3;
char_string_display downt (pixel_clock, pixel_count, line_count,
                                                                        down_text, "DOWN",
11'd370, 10'd660);

rectangle right_r (reset, pixel_clock, pixel_count, line_count,
                                          11'd549, 10'd592, 11'd699, 10'd752,
                                          right_border, right_inside);
defparam rightt.NCHAR = 5;
defparam rightt.NCHAR_BITS = 3;
char_string_display rightt (pixel_clock, pixel_count, line_count,
                                                                         right_text, "RIGHT",
11'd580, 10'd660);

rectangle left_r (reset, pixel_clock, pixel_count, line_count,
                                         11'd774, 10'd592, 11'd924, 10'd752,
                                         left_border, left_inside);
defparam leftt.NCHAR = 4;
defparam leftt.NCHAR_BITS = 3;
char_string_display leftt (pixel_clock, pixel_count, line_count,
                                                                                    left_text, "LEFT",
11'd811, 10'd660);

// generate enter, delete, and return buttons
wire select_border, enter_border, delete_border, return_border;
wire select_inside, enter_inside, delete_inside, return_inside;
wire select_text, enter_text, delete_text, return_text;

rectangle select_r (reset, pixel_clock, pixel_count, line_count,
                                                 11'd99, 10'd400, 11'd249, 10'd560,
                                                 select_border, select_inside);
defparam selectt.NCHAR = 6;
defparam selectt.NCHAR_BITS = 3;
char_string_display selectt (pixel_clock, pixel_count, line_count,
                                                                                     select_text, "SELECT",
11'd143, 10'd468);

rectangle enter_r (reset, pixel_clock, pixel_count, line_count,
                                                  11'd324, 10'd400, 11'd474, 10'd560,
                                                  enter_border, enter_inside);
defparam entert.NCHAR = 5;
defparam entert.NCHAR_BITS = 3;
char_string_display entert (pixel_clock, pixel_count, line_count,
                                                                                    enter_text, "ENTER",
11'd362, 10'd468);

rectangle delete_r (reset, pixel_clock, pixel_count, line_count,
                                                  11'd549, 10'd400, 11'd699, 10'd560,
                                                  delete_border, delete_inside);
defparam deletet.NCHAR = 6;
defparam deletet.NCHAR_BITS = 3;
char_string_display deletet (pixel_clock, pixel_count, line_count,
                                                                                     delete_text, "DELETE",
11'd564, 10'd468);

rectangle return_r (reset, pixel_clock, pixel_count, line_count,
                                                  11'd774, 10'd400, 11'd924, 10'd560,
                                                  return_border, return_inside);
defparam returnt.NCHAR = 6;
defparam returnt.NCHAR_BITS = 3;
char_string_display returnt (pixel_clock, pixel_count, line_count,
                                                                                     return_text, "RETURN",
11'd795, 10'd468);

wire[31:0] song_code_text;
wire [3:0] remone, remten, remhundred;
wire rfdx3;
numtochar num (reset, pixel_clock, song_code, remone, remten,
                                                remhundred, rfdx3, song_code_text);

char_string_display temp (pixel_clock, pixel_count,
                                                                             line_count, temp_text,
song_code_text, 11'd850, 10'd200);
```

```
// generating the text box where the song name entered so far are shown
wire [7:0] text_box_red, text_box_green, text_box_blue;
wire text_box_on;

text_box work_please (reset, pixel_clock, pixel_count, line_count,
                                        title,
                                                    text_box_on,
                                        text_box_red, text_box_green, text_box_blue);

// generating the key board where the user could select which letter
// to enter
wire [7:0] text_key_red, text_key_green, text_key_blue;
wire text_key_on;

text_key key (reset, pixel_clock, pixel_count, line_count, bg_on,
                              letter,

                              text_key_on,
                              text_key_red, text_key_green, text_key_blue);

reg [7:0] song_count;

always @ (posedge pixel_clock)          // assigning color values to
begin                                                                        // each
pixel
        if (reset | !on)
                begin
                        red <= 8'd0;
                        green <= 8'd0;
                        blue <= 8'd0;
                        song_count <= 8'd0;
                end
        else if (title_on | temp_text)
                begin
                        red <= 8'd0;
                        green <= 8'd0;
                        blue <= 8'd0;
                end
        else if (up_text | down_text | right_text | left_text |
                                select_text | enter_text | delete_text | return_text |
                                up_border | down_border | right_border | left_border |
                                select_border | enter_border | delete_border | return_border)
                begin
                        red <= 8'd0;
                        green <= 8'd0;
                        blue <= 8'd0;
                end
        else if (up_inside | down_inside | right_inside | left_inside |
                                select_inside | enter_inside | delete_inside | return_inside)
                begin
                        if (bg_on)
                                begin
                                        red <= 8'd255;
                                        green <= 8'b10000000;
                                        blue <= 8'd255;
                                end
                        else
                                begin
                                        red <= 8'd128;
                                        green <= 8'd128;
                                        blue <= 8'd128;
                                end
                end
        else if (text_box_on)
                begin
                        red <= text_box_red;
                        green <= text_box_green;
                        blue <= text_box_blue;
                end
        else if (text_key_on)
                begin
                        red <= text_key_red;
                        green <= text_key_green;
                        blue <= text_key_blue;
                end
        else if (bg_on)
                begin
                        red <= 8'd255;
                        green <= 8'd0;
                        blue <= 8'd255;
                end
        else
                begin
                        red <= 8'd0;
                        green <= 8'd0;
                        blue <= 8'd0;
                end

end


endmodule






`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    18:58:28 05/15/06
// Design Name:
// Module Name:    song_enter_cal
// Project Name:
// Target Device:
// Tool versions:
// Description:       This module takes user inputs determined by the step
//                                                      intepretation block, and outputs title_disp, which is
the
```

```
//                                                  title that the user had entered so far, and key_num,
which
//                                                  is the key that the user is currently highlighting, to
the
//                                                  display logic block to be shown on the screen, and it
also
//                                                  outputs the title_reg (the complete title), along with
the
//                                                  code for the song and a write_ready signal to SRAM
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
module song_enter_cal(reset, pixel_clock, pixel_count, line_count, on,
                                                        up, down, right, left, enter, delete, select,
                                                        title_disp, key_num, title_reg, song_code,
write_ready);

input reset, pixel_clock;
input [10:0] pixel_count;
input [9:0] line_count;
input on, up, down, right, left, enter, delete, select;

// to song_enter module to display text
output [255:0] title_disp;
reg [255:0] title_disp;

// testbench only
//output [15:0] title_disp;
//reg [15:0] title_disp;

// letter represented as 0 - 26, with 0 representing A, and 26 representing space
output [4:0] key_num;
reg [4:0] key_num;

// to audio and SRAM
output [255:0] title_reg;
reg [255:0] title_reg;
output [3:0] song_code;
reg [3:0] song_code;
output write_ready;
reg write_ready;

// testbench only
//output [15:0] title_reg;
//reg [15:0] title_reg;

reg [4:0] button_count;
reg [3:0] song_count;
reg [4:0] char_count;

always @ (posedge pixel_clock)
```

```
begin
        if (reset)
                begin
                        title_disp <= 0;
                        title_reg <= 0;
                        song_code <= 1; // song code start with 1
                        write_ready <= 0;
                        button_count <= 0;
                        key_num <= 0;
                        song_count <= 1;
                        char_count <= 0;
                end
        else if (!on)
                begin
                        title_disp <= 0;
                        title_reg <= 0;
                        write_ready <= 0;
                        button_count <= 0;
                        key_num <= 0;
                        char_count <= 0;
                end
//      else if (enter && (song_count < 15))            // enter is a pulsed signal that
//              begin
         // is high for one clock cycle only
//                      song_code <= song_count;                        // after integration --> cannot be
//                      song_count <= song_count + 1;                   // tracked for 15 frames
//                      title_reg <= title_disp;
//                      write_ready <= 1;
//                      title_disp <= 0;
//              end
//      else if (enter && (song_count == 15))
//              begin
//                      song_code <= song_count;
//                      song_count <= song_count;
//                      title_reg <= title_disp;
//                      write_ready <= 1;
//                      title_disp <= 0;
//              end
//      else if ((pixel_count == 0) && (line_count == 0)) // testbench only
        else if ((pixel_count == 0) && (line_count == 772))
                begin
                        if (button_count <= 15)                                         // only change
signals
                                button_count <= button_count + 1;  // every 15 frames
                        else
                                begin
                                        button_count <= 0;
                                        if (select && (char_count < 31))
//                                      else if (select)                // testbench
                                                begin
                                                        title_disp[255:248] <= title_disp[247:240];
// entering a
                                                        title_disp[247:240] <= title_disp[239:232];
// new letter
                                                        title_disp[239:232] <= title_disp[231:224];
// shift everything
```

```
                                                            title_disp[231:224] <= title_disp[223:216];
// else
                                                            title_disp[223:216] <= title_disp[215:208];
                                                            title_disp[215:208] <= title_disp[207:200];
                                                            title_disp[207:200] <= title_disp[199:192];
                                                            title_disp[199:192] <= title_disp[191:184];
                                                            title_disp[191:184] <= title_disp[183:176];
                                                            title_disp[183:176] <= title_disp[175:168];
                                                            title_disp[175:168] <= title_disp[167:160];
                                                            title_disp[167:160] <= title_disp[159:152];
                                                            title_disp[159:152] <= title_disp[151:144];
                                                            title_disp[151:144] <= title_disp[143:136];
                                                            title_disp[143:136] <= title_disp[135:128];
                                                            title_disp[135:128] <= title_disp[127:120];
                                                            title_disp[127:120] <= title_disp[119:112];
                                                            title_disp[119:112] <= title_disp[111:104];
                                                            title_disp[111:104] <= title_disp[103:96];
                                                            title_disp[103:96] <= title_disp[95:88];
                                                            title_disp[95:88] <= title_disp[87:80];
                                                            title_disp[87:80] <= title_disp[79:72];
                                                            title_disp[79:72] <= title_disp[71:64];
                                                            title_disp[71:64] <= title_disp[63:56];
                                                            title_disp[63:56] <= title_disp[55:48];
                                                            title_disp[55:48] <= title_disp[47:40];
                                                            title_disp[47:40] <= title_disp[39:32];
                                                            title_disp[39:32] <= title_disp[31:24];
                                                            title_disp[31:24] <= title_disp[23:16];
                                                            title_disp[23:16] <= title_disp[15:8];
                                                            title_disp[15:8] <= title_disp[7:0];
                                                            title_disp[7:0] <= key_num + 8'd65;
                                          // change key_num to ASCII
                                                            char_count <= char_count + 1;
                                                 // keep track of number of letters
                                                      end
                                                else if (select && (char_count == 31))
//                                              else if (select)      // testbench only
                                                      title_disp <= title_disp;
                                                else if (delete)
                                                      begin
                                                            char_count <= char_count - 1;
                                                      // deleting a letter
                                                            title_disp[255:248] <= 8'd0;
                                                      // shifting the existing
                                                            title_disp[247:240] <= title_disp[255:248];
                           // titles by 5 pages
                                                            title_disp[239:232] <= title_disp[247:240];
                                                            title_disp[231:224] <= title_disp[239:232];
                                                            title_disp[223:216] <= title_disp[231:224];
                                                            title_disp[215:208] <= title_disp[223:216];
                                                            title_disp[207:200] <= title_disp[215:208];
                                                            title_disp[199:192] <= title_disp[207:200];
                                                            title_disp[191:184] <= title_disp[199:192];
                                                            title_disp[183:176] <= title_disp[191:184];
                                                            title_disp[175:168] <= title_disp[183:176];
                                                            title_disp[167:160] <= title_disp[175:168];
                                                            title_disp[159:152] <= title_disp[167:160];
                                                            title_disp[151:144] <= title_disp[159:152];
                                                            title_disp[143:136] <= title_disp[151:144];
                                                            title_disp[135:128] <= title_disp[143:136];
                                                            title_disp[127:120] <= title_disp[135:128];
                                                            title_disp[119:112] <= title_disp[127:120];
                                                            title_disp[111:104] <= title_disp[119:112];
                                                            title_disp[103:96] <= title_disp[111:104];
                                                            title_disp[95:88] <= title_disp[103:96];
                                                            title_disp[87:80] <= title_disp[95:88];
                                                            title_disp[79:72] <= title_disp[87:80];
                                                            title_disp[71:64] <= title_disp[79:72];
                                                            title_disp[63:56] <= title_disp[71:64];
                                                            title_disp[55:48] <= title_disp[63:56];
                                                            title_disp[47:40] <= title_disp[55:48];
                                                            title_disp[39:32] <= title_disp[47:40];
                                                            title_disp[15:8] <= 0;
                                                            title_disp[31:24] <= title_disp[39:32];
                                                            title_disp[23:16] <= title_disp[31:24];
                                                            title_disp[23:16] <= title_disp[31:24];
                                                            title_disp[23:16] <= title_disp[31:24];
                                                            title_disp[15:8] <= title_disp[23:16];
                                                            title_disp[7:0] <= title_disp[15:8];
                                                      end
                                                else if (up && (key_num >= 9))
                           // calculates which
                                                      key_num <= key_num - 9;
                                           //        letter is currently
                                                else if (down && (key_num <= 17))
                           // highlighted from the
                                                      key_num <= key_num + 9;
                                                // keyboard configuration
                                                else if (right && (key_num < 26))
                           // and user inputs
                                                      key_num <= key_num + 1;
                                                else if (left && (key_num > 0))
                                                      key_num <= key_num - 1;
                       // for testing, take out after integration                                          // testing
                       with button

                                                                           // inputs, needs to
                                                else if (enter && (song_count < 15))
                           // add the changing every
                                                      begin
                                                                // 15 frames constraints
                                                            song_code <= song_count;
                                                            song_count <= song_count + 1;
                                                            title_reg <= title_disp;
                                                            write_ready <= 1;
                                                            title_disp <= 0;
                                                      end
                                                else if (enter && (song_count == 15))
                                                      begin
                                                            song_code <= song_count;
                                                            song_count <= song_count;
                                                            title_reg <= title_disp;
                                                            write_ready <= 1;
```

```
                                                                                        title_disp <= 0;
                                                                            end
// for testing, take out after integration

                                                            else
                                                                            key_num <= key_num;
                                                    end
                                    end
                    else
                                    write_ready <= 0;


end
endmodule
```

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    19:46:14 05/13/06
// Design Name:
// Module Name:    song_sel_box
// Project Name:
// Target Device:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
module song_sel_box(reset, pixel_clock, pixel_count, line_count, bg_on,
                                                                                up, down, enter,
                                                                                sel_box_on, red, green, blue, song_code,
code_ready);

input reset, pixel_clock;
input [10:0] pixel_count;
input [9:0] line_count;
input bg_on, up, down, enter;

output sel_box_on;
output [7:0] red, green, blue;
reg sel_box_on;
reg [7:0] red, green, blue;
output [3:0] song_code;
reg [3:0] song_code;
```

```
output code_ready;
reg code_ready;

wire sel_border, sel_inside;
wire [4:0] sel_text;

rectangle sel_box (reset, pixel_clock, pixel_count, line_count,
                                                                        11'd149, 10'd199, 11'd873, 10'd385,
                                                              sel_border, sel_inside);

reg [3:0] song1, song2, song3, song4, song5;
reg [3:0] current_song;

wire [255:0] song_title1_m, song_title2_m, song_title3_m, song_title4_m, song_title5_m;
wire ready;
reg [255:0] song_title1, song_title2, song_title3, song_title4, song_title5;

//get_title title1 (reset, pixel_clock, song1, song2, song3, song4, song5,
//                                                                      song_title1_m, song_title2_m, song_title3_m,
song_title4_m,
//                                                                      song_title5_m, ready);
//
defparam titlec1.NCHAR = 32;
defparam titlec1.NCHAR_BITS = 6;
char_string_display titlec1 (pixel_clock, pixel_count, line_count,
                                                                                                  sel_text[0], song_title1,
11'd249, 10'd211);
defparam titlec2.NCHAR = 32;
defparam titlec2.NCHAR_BITS = 6;
char_string_display titlec2 (pixel_clock, pixel_count, line_count,
                                                                                                  sel_text[1], song_title2,
11'd249, 10'd247);
defparam titlec3.NCHAR = 32;
defparam titlec3.NCHAR_BITS = 6;
char_string_display titlec3 (pixel_clock, pixel_count, line_count,
                                                                                                  sel_text[2], song_title3,
11'd249, 10'd283);
defparam titlec4.NCHAR = 32;
defparam titlec4.NCHAR_BITS = 6;
char_string_display titlec4 (pixel_clock, pixel_count, line_count,
                                                                                                  sel_text[3], song_title4,
11'd249, 10'd319);
defparam titlec5.NCHAR = 32;
defparam titlec5.NCHAR_BITS = 6;
char_string_display titlec5 (pixel_clock, pixel_count, line_count,
                                                                                                  sel_text[4], song_title5,
11'd249, 10'd355);

reg [4:0] count;

always @ (posedge pixel_clock)
begin
        if (reset)
                begin
                        red <= 8'd0;
                        green <= 8'd0;
```

```
                    blue <= 8'd0;
                    sel_box_on <= 0;
                    song1 <= 0;
                    song2 <= 1;
                    song3 <= 2;
                    song4 <= 3;
                    song5 <= 4;
                    current_song <= 0;
                    count <= 0;
                    song_code <= 0;
                    code_ready <= 0;
          end
if (ready)
          begin
                    song_title1 <= song_title1_m;
                    song_title2 <= song_title2_m;
                    song_title3 <= song_title3_m;
                    song_title4 <= song_title4_m;
                    song_title5 <= song_title5_m;
          end
else if (enter)
          begin
                    song_code <= current_song;
                    code_ready <= 1;
          end
else if ((pixel_count == 0) && (line_count == 772))
          begin
                    if (count <= 15)
                              count <= count + 1;
                    else
                              begin
                                        count <= 0;
                                        if (up && (current_song == song1) && (song1 > 0))
                                                  begin
                                                            song1 <= song1 - 1;
                                                            song2 <= song2 - 1;
                                                            song3 <= song3 - 1;
                                                            song4 <= song4 - 1;
                                                            song5 <= song5 - 1;
                                                            current_song <= song1 - 1;
                                                  end
                                        else if (up)
                                                  begin
                                                            if ((current_song == song1) ||
                                                                       (current_song == song2))
                                                                      current_song <= song1;
                                                            else if (current_song == song3)
                                                                      current_song <= song2;
                                                            else if (current_song == song4)
                                                                      current_song <= song3;
                                                            else
                                                                      current_song <= song4;
                                                  end
                                        else if (down && (current_song == song5) && (song5 <255))
                                                  begin
                                                            song1 <= song1 + 1;
```

```
                        song2 <= song2 + 1;
                        song3 <= song3 + 1;
                        song4 <= song4 + 1;
                        song5 <= song5 + 1;
                        current_song <= song5 + 1;
                    end
                else if (down)
                    begin
                        if (current_song == song1)
                            current_song <= song2;
                        else if (current_song == song2)
                            current_song <= song3;
                        else if (current_song == song3)
                            current_song <= song4;
                        else
                            current_song <= song5;
                    end
            end
    end
else if (sel_inside)
    begin
        sel_box_on <= 1;
        if (sel_text != 0)
            begin
                if (((current_song == song1) && sel_text[0]) ||
                     ((current_song == song2) && sel_text[1]) ||
                     ((current_song == song3) && sel_text[2]) ||
                     ((current_song == song4) && sel_text[3]) ||
                     ((current_song == song5) && sel_text[4]))
                    begin
                        red <= 8'd255;
                        green <= 8'd255;
                        blue <= 8'd255;
                    end
                else
                    begin
                        red <= 8'd0;
                        green <= 8'd0;
                        blue <= 8'd0;
                    end
            end
        else if (bg_on)
            begin
                red <= 8'd255;
                green <= 8'd128;
                blue <= 8'd255;
            end
        else
            begin
                red <= 8'd128;
                green <= 8'd128;
                blue <= 8'd128;
            end
    end
else if (sel_border)
    begin
```

```
                        sel_box_on <= 1;
                        red <= 8'd0;
                        green <= 8'd0;
                        blue <= 8'd0;
                end
        else
                begin
                        red <= 8'd0;
                        green <= 8'd0;
                        blue <= 8'd0;
                        sel_box_on <= 0;
                end
end

endmodule
```

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    16:38:14 05/15/06
// Design Name:
// Module Name:    song_sel_cal
// Project Name:
// Target Device:
// Tool versions:
// Description:      Given user inputs (up and down and select), calculates which
//                                                  songs should be displayed, keep track of which song
is the
//                                                  cursor on, and gets the appropriate song titles from
SRAM
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
module song_sel_cal(reset, pixel_clock, pixel_count, line_count, on,
                                                  up, down, enter,
                                                  song_title_1, song_title_2, song_title_3,
song_title_4,
                                                  song_title_5, data_ready,
                                                  song1, song2, song3, song4, song5, read_ready,
                                                  current_pos, selected_song,
                                                  latched_song_title_1, latched_song_title_2,
                                                  latched_song_title_3, latched_song_title_4,
                                                  latched_song_title_5);
```

```
input reset, pixel_clock;
input [10:0] pixel_count;
input [9:0] line_count;
input on;
input up, down, enter;

// data from SRAM, will be glicthy until it is ready;
input [255:0] song_title_1, song_title_2, song_title_3, song_title_4, song_title_5;
//input [39:0] song_title_1, song_title_2, song_title_3, song_title_4, song_title_5;

// pulsed singal indicating that the song_titles have become stable
input data_ready;

output [3:0] song1, song2, song3, song4, song5;
reg [3:0] song1, song2, song3, song4, song5;
output read_ready; // pulsed signal to SRAM indicating the start of reading
reg read_ready;

// output to the song_selection module, which display the slide, to highlight
// the song --> will change with up and down buttons
output [3:0] current_pos;

// output to the audio portion of the project, value determined when user
// press enter
output [3:0] selected_song;
reg [3:0] current_pos, selected_song;

output [255:0] latched_song_title_1, latched_song_title_2, latched_song_title_3,
                                        latched_song_title_4, latched_song_title_5;
reg [255:0] latched_song_title_1, latched_song_title_2, latched_song_title_3,
                                        latched_song_title_4, latched_song_title_5;

reg [4:0] count;
reg [3:0] current_song;
//output first_on;          // testbench only
reg first_on;

always @ (posedge pixel_clock)
begin
        if (reset | !on)
                begin
                        song1 <= 0;
                        song2 <= 1;
                        song3 <= 2;
                        song4 <= 3;
                        song5 <= 4;
                        current_song <= 0;
                        selected_song <= 0;
                        latched_song_title_1 <= 0;
                        latched_song_title_2 <= 0;
                        latched_song_title_3 <= 0;
                        latched_song_title_4 <= 0;
                        latched_song_title_5 <= 0;
                        count <= 0;
                        current_pos <= 0;
                        read_ready <= 0;
```

```
                                first_on <= 0;
                        end
                else if (first_on == 0)                 // sends a read_ready pulse when
                        begin                                                                   // module first activated
                                first_on <= 1;
                                read_ready <= 1;
                        end
                else if (read_ready)
                        read_ready <= 0;
                else if (data_ready)                    // loading the new titles
                        begin
                                latched_song_title_1 <= song_title_1;
                                latched_song_title_2 <= song_title_2;
                                latched_song_title_3 <= song_title_3;
                                latched_song_title_4 <= song_title_4;
                                latched_song_title_5 <= song_title_5;
                        end
                else if (enter)
                                selected_song <= current_song;
//              else if ((pixel_count == 0) && (line_count == 5))               // testbench only
                else if ((pixel_count == 0) && (line_count == 772))  // adding constraints of
                        begin
                                                // changing every 15 screen
                                if (count <= 15)
                          // refreshes
                                        count <= count + 1;
                                else
                                        begin
                                                count <= 0;
                                                if (up && (current_song == song1))                      //
moving cursors and finding
                                                        begin
                                                 // new songs
                                                                if (song1 > 0)
                                                                        begin
                                                                                song1 <= song1 - 1;
                                                                                song2 <= song2 - 1;
                                                                                song3 <= song3 - 1;
                                                                                song4 <= song4 - 1;
                                                                                song5 <= song5 - 1;
                                                                                current_song <= song1 -
1;

                                                                                current_pos <= 0;
                                                                                read_ready <= 1;
                                                                        end
                                                                else
                                                                        begin
                                                                                song1 <= song1;
                                                                                song2 <= song2;
                                                                                song3 <= song3;
                                                                                song4 <= song4;
                                                                                song5 <= song5;
                                                                                current_song <= song1;
                                                                                current_pos <= 0;
                                                                                read_ready <= 0;
                                                                        end
```

```
                    end
                else if (up)
                    begin
                        current_pos <= current_pos - 1;
                        read_ready <= 0;
                        if ((current_song == song1) ||
                             (current_song == song2))
                            current_song <= song1;
                        else if (current_song == song3)
                            current_song <= song2;
                        else if (current_song == song4)
                            current_song <= song3;
                        else
                            current_song <= song4;
                    end
                else if (down && (current_song == song5))
                    begin
                        if (song5 < 15)
                            begin
                                song1 <= song1 + 1;
                                song2 <= song2 + 1;
                                song3 <= song3 + 1;
                                song4 <= song4 + 1;
                                song5 <= song5 + 1;
                                current_song <= song5 +
1;

                                current_pos <= 4;
                                read_ready <= 1;
                            end
                        else
                            begin
                                song1 <= song1;
                                song2 <= song2;
                                song3 <= song3;
                                song4 <= song4;
                                song5 <= song5;
                                current_song <= song5;
                                current_pos <= 4;
                                read_ready <= 0;
                            end
                    end
                else if (down)
                    begin
                        read_ready <= 0;
                        current_pos <= current_pos + 1;
                        if (current_song == song1)
                            current_song <= song2;
                        else if (current_song == song2)
                            current_song <= song3;
                        else if (current_song == song3)
                            current_song <= song4;
                        else
                            current_song <= song5;
                    end
            end
        end
```

```
		else
			begin
				latched_song_title_1 <= latched_song_title_1;
				latched_song_title_2 <= latched_song_title_2;
				latched_song_title_3 <= latched_song_title_3;
				latched_song_title_4 <= latched_song_title_4;
				latched_song_title_5 <= latched_song_title_5;
				current_pos <= current_pos;
				read_ready <= 0;
			end

end

endmodule
```

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    17:03:57 05/13/06
// Design Name:
// Module Name:    song_selection
// Project Name:
// Target Device:
// Tool versions:
// Description:      Draws the song selection screen
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
module song_selection(reset, pixel_clock, pixel_count, line_count, on,
                                                  song_title_1, song_title_2, song_title_3,
song_title_4,
                                                  song_title_5, current_song,
                                                  red, green, blue);

input reset, pixel_clock;
input [10:0] pixel_count;
input [9:0] line_count;
input on;
input [255:0] song_title_1, song_title_2, song_title_3, song_title_4, song_title_5;
// 0-4 presenting which song is currently being high-lighted
input [3:0] current_song;

output [7:0] red, green, blue;
reg [7:0] red, green, blue;

// draws the background
wire bg_on;
vga_romdisp bg (pixel_clock, pixel_count, line_count, pixel_clock, bg_on);

// write the title of the slide
wire title_on;
defparam title.NCHAR = 13;
defparam title.NCHAR_BITS = 4;
char_string_display title (pixel_clock, pixel_count, line_count,
                                                               title_on, "SELECT A
SONG", 11'd408, 10'd50);

// draw the buttons
wire up_border, down_border, enter_border, return_border;
wire up_inside, down_inside, enter_inside, return_inside;
wire up_text, down_text, enter_text, return_text;

rectangle up_r (reset, pixel_clock, pixel_count, line_count,
                              11'd99, 10'd450, 11'd249, 10'd650,
                              up_border, up_inside);
defparam upt.NCHAR = 2;
defparam upt.NCHAR_BITS = 2;
char_string_display upt (pixel_clock, pixel_count, line_count,
                                                         up_text, "UP", 11'd165, 10'd540);

rectangle down_r (reset, pixel_clock, pixel_count, line_count,
                                   11'd324, 10'd450, 11'd474, 10'd650,
                                        down_border, down_inside);
defparam downt.NCHAR = 4;
defparam downt.NCHAR_BITS = 3;
char_string_display downt (pixel_clock, pixel_count, line_count,
                                                                            down_text, "DOWN",
11'd370, 10'd540);

rectangle enter_r (reset, pixel_clock, pixel_count, line_count,
                                        11'd549, 10'd450, 11'd699, 10'd650,
                                        enter_border, enter_inside);
defparam entert.NCHAR = 5;
defparam entert.NCHAR_BITS = 3;
char_string_display entert (pixel_clock, pixel_count, line_count,
                                                                            enter_text, "ENTER",
11'd580, 10'd540);

rectangle return (reset, pixel_clock, pixel_count, line_count,
                                        11'd774, 10'd450, 11'd924, 10'd650,
                                        return_border, return_inside);
defparam returnt.NCHAR = 6;
defparam returnt.NCHAR_BITS = 3;
char_string_display returnt (pixel_clock, pixel_count, line_count,
                                                                            return_text, "RETURN",
11'd795, 10'd540);

// draws the title of the 5 songs that are displayed
wire sel_border, sel_inside;
wire [4:0] sel_text;

defparam titlec1.NCHAR = 32;
defparam titlec1.NCHAR_BITS = 6;
char_string_display titlec1 (pixel_clock, pixel_count, line_count,
                                                                            sel_text[0], song_title_1,
11'd249, 10'd211);
defparam titlec2.NCHAR = 32;
defparam titlec2.NCHAR_BITS = 6;
char_string_display titlec2 (pixel_clock, pixel_count, line_count,
                                                                            sel_text[1], song_title_2,
11'd249, 10'd247);
defparam titlec3.NCHAR = 32;
defparam titlec3.NCHAR_BITS = 6;
char_string_display titlec3 (pixel_clock, pixel_count, line_count,
                                                                            sel_text[2], song_title_3,
11'd249, 10'd283);
defparam titlec4.NCHAR = 32;
defparam titlec4.NCHAR_BITS = 6;
char_string_display titlec4 (pixel_clock, pixel_count, line_count,
                                                                            sel_text[3], song_title_4,
11'd249, 10'd319);
defparam titlec5.NCHAR = 32;
defparam titlec5.NCHAR_BITS = 6;
char_string_display titlec5 (pixel_clock, pixel_count, line_count,
                                                                            sel_text[4], song_title_5,
11'd249, 10'd355);

// draws the song title box
```

```
rectangle sel_box (reset, pixel_clock, pixel_count, line_count,
                                        11'd149, 10'd199, 11'd873, 10'd385,
                                   sel_border, sel_inside);


always @ (posedge pixel_clock)           // assign the color values based on
begin                                                                     // pixel
numbers
        if (reset | (!on))
                begin
                        red <= 0;
                        green <= 0;
                        blue <= 0;
                end
        else if (title_on)
                begin
                        red <= 8'd0;
                        green <= 8'd0;
                        blue <= 8'd0;
                end
        else if (((current_song == 0) && sel_text[0]) ||
                                ((current_song == 1) && sel_text[1]) ||
                                ((current_song == 2) && sel_text[2]) ||
                                ((current_song == 3) && sel_text[3]) ||
                                ((current_song == 4) && sel_text[4]))
                begin
                        red <= 8'd255;
                        green <= 8'd255;
                        blue <= 8'd255;
                end
        else if (sel_text != 5'd0)
                begin
                        red <= 8'd0;
                        green <= 8'd0;
                        blue <= 8'd0;
                end
        else if (up_text | down_text | return_text | enter_text |
                                up_border | down_border | return_border |
                                enter_border | sel_border)
                begin
                        red <= 8'd0;
                        green <= 8'd0;
                        blue <= 8'd0;
                end
        else if (up_inside | down_inside | enter_inside | return_inside | sel_inside)
                begin
                        if (bg_on)
                                begin
                                        red <= 8'd255;
                                        green <= 8'b10000000;
                                        blue <= 8'd255;
                                end
                        else
                                begin
                                        red <= 8'd128;
                                        green <= 8'd128;
                                        blue <= 8'd128;
                                end
                        end
                else if (bg_on)
                        begin
                                red <= 8'd255;
                                green <= 8'd0;
                                blue <= 8'd255;
                        end
                else
                        begin
                                red <= 8'd0;
                                green <= 8'd0;
                                blue <= 8'd0;
                        end
end
endmodule
```

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    01:13:03 05/15/06
// Design Name:
// Module Name:    song_text
// Project Name:
// Target Device:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
module song_text(reset, pixel_clock, pixel_count, line_count,
                                        enter, add_letter, delete, letter,
                                        text_box_on, title_ready,
                                        red, green, blue, song_title);

input reset, pixel_clock;
input [10:0] pixel_count;
input [9:0] line_count;
input enter, add_letter, delete;
input [7:0] letter;

output text_box_on;
reg text_box_on;
```

```
output title_ready;
reg title_ready;
output [7:0] red, green, blue;
reg [7:0] red, green, blue;

output [255:0] song_title;
reg [255:0] song_title;


wire pixel_clock;
wire [10:0] pixel_count;
wire [9:0] line_count;

//output box_border, box_inside;
wire box_border, box_inside;
rectangle rec (0, pixel_clock, pixel_count, line_count,
                                        11'd100, 10'd300, 11'd900, 10'd400,
                                        box_border, box_inside);

always @ (posedge pixel_clock)
begin
        if (reset)
                begin
                        red <= 0;
                        green <= 0;
                        blue <= 0;
                        text_box_on <= 0;
                        title_ready <= 0;
                        song_title <= 0;
                end
        else if (box_border || box_inside)
                begin
                        red <= 8'b11111111;
                        green <= 8'b11111111;
                        blue <= 8'b11111111;
                        text_box_on <= 1;
                end
        else
                begin
                        red <= 8'd0;
                        green <= 8'd0;
                        blue <= 8'd0;
                        text_box_on <= 0;
                end
end



endmodule





`timescale 1ns / 1ps
```

```
////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    12:59:47 03/21/06
// Design Name:
// Module Name:    sync_signal_generator
// Project Name:
// Target Device:
// Tool versions:
// Description:       Generate the sync and blank signals for VGA display
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
module sync_signal_generator (reset, clk, hblank, vblank, hsync, vsync, pixel_count, line_count);

input reset, clk;
output hblank, vblank, hsync, vsync;
output [10:0] pixel_count;
output [9:0] line_count;
reg hblank, vblank, hsync, vsync;
reg [10:0] pixel_count;
reg [9:0] line_count;


/*parameter hblank_high = 640;                    // for 640x480
parameter h_front_porch = 16;
parameter hsync_low = 96;
parameter h_back_porch = 48;
parameter h_period = 800;
parameter vblank_high = 480;
parameter v_front_porch = 11;
parameter vsync_low = 2;
parameter v_back_porch = 32;
parameter v_period = 525;*/


// testbench purposes only
/*parameter hblank_high = 6;
parameter h_front_porch = 1;
parameter hsync_low = 3;
parameter h_back_porch = 2;
parameter h_period = 12;
parameter vblank_high = 5;
parameter v_front_porch = 1;
parameter vsync_low = 2;
parameter v_back_porch = 3;
parameter v_period = 11;
*/

parameter hblank_high = 1024;                                    // for 1024x768
parameter h_front_porch = 24;
parameter hsync_low = 136;
parameter h_back_porch = 160;
parameter h_period = 1344;
parameter vblank_high = 768;
parameter v_front_porch = 3;
parameter vsync_low = 6;
parameter v_back_porch = 29;
parameter v_period = 806;

always @ (posedge clk)
begin
        if (reset)
        begin
                hsync <= 1;
                vsync <= 1;
                hblank <= 1;
                vblank <= 1;
                pixel_count <= 0;
                line_count <= 0;
        end
        else if (pixel_count < (hblank_high - 1))
//      else if (pixel_count < 639)  // switch to this upon implementation
        begin                                           // should save space
                hblank <= 1;
                hsync <= 1;
                pixel_count <= pixel_count + 1;
        end
        else if (pixel_count < (hblank_high + h_front_porch - 1))
//      else if (pixel_count < 655)
        begin
                hblank <= 0;
                hsync <= 1;
                pixel_count <= pixel_count + 1;
        end
        else if (pixel_count < (hblank_high + h_front_porch + hsync_low - 1))
//      else if (pixel_count < 751)
        begin
                hblank <= 0;
                hsync <= 0;
                pixel_count <= pixel_count + 1;
        end
        else if (pixel_count < (h_period - 1))
//      else if (pixel_count        < 199)
        begin
                hblank <= 0;
                hsync <= 1;
                pixel_count <= pixel_count + 1;
        end
        else
        begin
                pixel_count <= 0;
                hblank <= 1;
                hsync <= 1;
//              if (line_count < 479)
                if (line_count < (vblank_high - 1))
```

```
			begin
				vsync <= 1;
				vblank <= 1;
				line_count <= line_count + 1;
			end
			else if (line_count < (vblank_high + v_front_porch - 1))
//			else if (line_count < 490)
			begin
				vsync <= 1;
				vblank <= 0;
				line_count <= line_count + 1;

			end
			else if (line_count < (vblank_high + v_front_porch + vsync_low - 1))
//			else if (line_count < 492)
			begin
				vsync <= 0;
				vblank <= 0;
				line_count <= line_count + 1;
			end
			else if (line_count < (v_period - 1))
//			else if (line_count < 524)
			begin
				vsync <= 1;
				vblank <= 0;
				line_count <= line_count + 1;
			end
			else
			begin
				line_count <= 0;
				vsync <= 1;
				vblank <= 1;
			end
		end
end

endmodule
```

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    22:22:38 05/14/06
// Design Name:
// Module Name:    text_box
// Project Name:
// Target Device:
// Tool versions:
// Description:      generate the text_box where the title that user enter so
//                                                         far is shown
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
module text_box (reset, pixel_clock, pixel_count, line_count,
                                                    title,
                                                    text_box_on,
                                                    red, green, blue);

input reset, pixel_clock;
input [10:0] pixel_count;
input [9:0] line_count;
input [255:0] title;

output text_box_on;
reg text_box_on;
output [7:0] red, green, blue;
reg [7:0] red, green, blue;

// draw the text box
wire text_box_border, text_box_inside;
rectangle show (reset, pixel_clock, pixel_count, line_count,
                                                   11'd149, 10'd300, 11'd874, 10'd360,
                                                   text_box_border, text_box_inside);

// display text
wire song_title_on;
defparam ch.NCHAR = 32;
defparam ch.NCHAR_BITS = 5;
char_string_display ch (pixel_clock, pixel_count, line_count,
                                                                 song_title_on, title, 11'd255, 10'd318);


always @ (posedge pixel_clock)      // assigning color outputs based on
begin                                                                                  // pixels
          if (reset)
                    begin
                              text_box_on <= 0;
                              red <= 0;
                              green <= 0;
                              blue <= 0;
                    end
          else if (song_title_on | text_box_border)
                    begin
                              red <= 8'd0;
                              green <= 8'd0;
                              blue <= 8'd0;
                              text_box_on <= 1;
                    end
          else if (text_box_inside)
                    begin
```

```
                red <= 8'd255;
                green <= 8'd255;
                blue <= 8'd255;
                text_box_on <= 1;
            end
        else
            begin
                red <= 8'd0;
                green <= 8'd0;
                blue <= 8'd0;
                text_box_on <= 0;
            end
end

endmodule
```

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    17:18:56 05/14/06
// Design Name:
// Module Name:    text_key
// Project Name:
// Target Device:
// Tool versions:
// Description:       draws the keyboard, and highlight the key that the user selected
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
module text_key(reset, pixel_clock, pixel_count, line_count,
                        bg_on, key_num,
                        text_key_on,
                        red, green, blue);

input reset, pixel_clock;
input [10:0] pixel_count, line_count;
input bg_on;
input [4:0] key_num;

output text_key_on;
reg text_key_on;

output [7:0] red, green, blue;
reg [7:0] red, green, blue;
```

```
// key_num of 0 represent letter A, 1 represent letter B...
// 28 represent letter Z, and 26 represent a space
//reg [7:0] key_num;
reg [4:0] letter;
reg [4:0] letter_v;
reg [2:0] letter_h;

// draws the border of the key, letter_v and letter_h are updated as
// pixel_count and line_count change
wire key_border;
wire key_inside;
rectangle key1 (reset, pixel_clock, pixel_count, line_count,
                                        letter_v*60+11'd99, letter_h*60+10'd99,
                                        letter_v*60+11'd139, letter_h*60+10'd139,
                                        key_border, key_inside);

// draws the key
wire char_on;
defparam char.NCHAR = 1;
defparam char.NCHAR_BITS = 1;

// letter gets updated as the pixel_count and line_count change
char_string_display char (pixel_clock, pixel_count, line_count,
                                                                char_on, letter+8'd65,
letter_v*60+11'd111, letter_h*60+10'd107);

//reg [4:0] move_count;

always @ (posedge pixel_clock)
begin
        if (reset)
                begin
                        red <= 8'd0;
                        green <= 8'd0;
                        blue <= 8'd0;
                        text_key_on <= 0;
                        letter_h <= 0;
                        letter_v <= 0;
                end
        else if (line_count == letter_h*60+10'd140) // update letter_h and letter
                begin
                 // after one key finishes
                        text_key_on <= 1;
                        if (letter_h < 2)
                                begin
                                        letter_h <= letter_h + 1;
                                        letter <= letter + 9;
                                end
                        else
                                begin
                                        letter_h <= 0;
                                        letter <= 0;
                                end
                end
        else if (pixel_count == letter_v*60+11'd139) // update letter_v and letter
```

```
                    begin
                         // after one key finsihes
                        if ((line_count >= letter_h*60+10'd99) &&
                             (line_count <= letter_h*60+10'd139))
                            begin
                                text_key_on <= 1;
                                if (letter_v < 8)
                                    begin
                                        letter_v <= letter_v + 1;
                                        letter <= letter + 1;
                                    end
                                else
                                    begin
                                        letter_v <= 0;
                                        letter <= letter - 8;
                                    end
                            end
                        else
                            text_key_on <= 0;
                    end
                else if (key_border)                                                    //
        assigning the colors based on
                    begin
                     // the pixels
                        text_key_on <= 1;
                        red <= 8'd0;
                        green <= 8'd0;
                        blue <= 8'd0;
                    end
                else if (char_on)
                    begin
                        text_key_on <= 1;
                        if (key_num == letter)
                            begin
                                red <= 8'd0;
                                green <= 8'd0;
                                blue <= 8'd255;
                            end
                        else
                            begin
                                red <= 8'd255;
                                green <= 8'd255;
                                blue <= 8'd255;
                            end
                    end
                else if (key_inside)
                    begin
                        if (0 == letter)
                            begin
                                text_key_on <= 1;
                                red <= 8'd0;
                                green <= 8'd0;
                                blue <= 8'd255;
                            end
                        if (bg_on)
                            begin
```

```
                                                  text_key_on <= 1;
                                                  red <= 8'd255;
                                                  green <= 8'd128;
                                                  blue <= 8'd255;
                                        end
                              else
                                        begin
                                                  text_key_on <= 1;
                                                  red <= 8'd128;
                                                  green <= 8'd128;
                                                  blue <= 8'd128;
                                        end
                    end
          else
                    begin
                              text_key_on <= 0;
                              red <= 8'd255;
                              green <= 8'd255;
                              blue <= 8'd255;
                    end
end

endmodule
```

```
//
// File:   vga_romdisp.v
// Date:   14-Nov-05
// Author: I. Chuang                              (modified by Helen Liang)
//
// Example demonstrating display of image from ROM on 1024x768 VGA
//
// This module reads data from ROM and creates the proper 8-bit pixel
// value for a pixel position defined by (hcount,vcount).  Note that
// there is a one cycle delay in reading from memory, so we may
// have a single pixel offset error here, to be fixed.  But the displayed
// image looks respectable nevertheless.
//
// To create the image ROM, use the Xilinx IP tools to generate a
// single port block memory, and load in an initial value file (*.coe)
// for your image.  This ROM should have width 1
// and depth 196608 (512x384).
//
// The COE file may be generated from an 8-bit PGM format image file
// using pgm2coe.py, or using your own tool.  Read the Xilinx documentation
// for more about COE files.

module vga_romdisp(clk,hcount,vcount,pix_clk,pixel);

   input clk;                    // video clock
   input [10:0] hcount;          // current x,y location of pixel
   input [9:0] vcount;
   input pix_clk;   // pixel clock
   output pixel;     // 1 if it's background, 0 if it's body

   // the memory address is hcount/2 + vcount/2 * 512
   // (4 pixels per memory location, since image is 512x384, and
   // display is 1024x768).
/*
   reg [17:0]         raddr;
   always @(posedge clk)
     raddr <= (hcount==0 & vcount==0) ? 0
                  : (hcount==0 & pix_clk & ~vcount[0]) ? raddr + 512 : raddr;

   wire [17:0]       addr = {9'b0,hcount[10:2]} + raddr[17:0];
*/

           wire [17:0] addr = (vcount/2)*512 + (hcount/2);

   reg [17:0]         addr_reg;
   always @(posedge clk) addr_reg <= addr;

   // instantiate the image rom
   flower2_bw_320_240 imgrom(addr_reg[17:0],clk,pixel);

endmodule // vga_romdisp
```

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    18:19:24 04/13/06
// Design Name:
// Module Name:    white_key
// Project Name:
// Target Device:
// Tool versions:
// Description:      draw a white key given the row and col number of the key
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
module white_key(reset, pixel_clock, keyrow, keycol, pixel_count, line_count,
                                                      border, inside);

input reset, pixel_clock;
input [1:0] keyrow;
input [2:0] keycol;
input [10:0] pixel_count;
input [9:0] line_count;

output border, inside;

reg [10:0] top_corner_pixel, bottom_corner_pixel;
reg [9:0] top_corner_line, bottom_corner_line;

// instantiates a rectangle at the appropriate position
rectangle k0 (reset, pixel_clock, pixel_count, line_count,
                                    top_corner_pixel, top_corner_line,
                                    bottom_corner_pixel, bottom_corner_line,
                                    border, inside);

always @ (posedge pixel_clock)
begin
        if (reset)
                begin
                        top_corner_pixel <= 0;
                        top_corner_line <= 0;
                        bottom_corner_pixel <= 0;
                        bottom_corner_line <= 0;
                end
        else
                begin
                        top_corner_pixel <= 120*keycol + 1;
                        bottom_corner_pixel <= 120*(keycol + 1) + 1;
                        top_corner_line <= 256*keyrow;
                        bottom_corner_line <= 256*(keyrow + 1) - 1;
                end
end


endmodule
```

Song Memory Module

```
/*

  Simple test of Song Memory
  Uses Button 0 and Up to Read. (Output on LED lights)
  Uses Button 1,2,3 for writing.
  Song and note addresses are specified using switches.
  Note and duration data is specified using switches.

 */

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

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

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

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

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

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

               clock_feedback_out, clock_feedback_in,

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

               rs232_txd, rs232_rxd, rs232_rts, rs232_cts,

               mouse_clock, mouse_data, keyboard_clock, keyboard_data,

               clock_27mhz, clock1, clock2,

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

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

               switch,

               led,

               user1, user2, user3, user4,

               daughtercard,

               systemace_data, systemace_address, systemace_ce_b,
               systemace_we_b, systemace_oe_b, systemace_irq, systemace_mpbrdy,
                  analyzer1_data, analyzer1_clock,
                  analyzer2_data, analyzer2_clock,
                  analyzer3_data, analyzer3_clock,
                  analyzer4_data, analyzer4_clock);

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

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

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

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

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

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

input  clock_feedback_in;
output clock_feedback_out;

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

output rs232_txd, rs232_rts;
input  rs232_rxd, rs232_cts;

input  mouse_clock, mouse_data, keyboard_clock, keyboard_data;

input  clock_27mhz, clock1, clock2;

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

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

```
output [7:0] led;

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

inout [43:0] daughtercard;

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

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

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

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

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

// Video Input
assign tv_in_i2c_clock = 1'b0;
assign tv_in_fifo_read = 1'b0;
assign tv_in_fifo_clock = 1'b0;
assign tv_in_iso = 1'b0;
assign tv_in_reset_b = 1'b0;
assign tv_in_clock = 1'b0;
assign tv_in_i2c_data = 1'bZ;

// SRAMs
assign ram0_data = 36'hZ;
assign ram0_address = 19'h0;
assign ram0_adv_ld = 1'b0;
assign ram0_clk = 1'b0;
assign ram0_cen_b = 1'b1;
assign ram0_ce_b = 1'b1;
assign ram0_oe_b = 1'b1;
```

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

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

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

   // LED Displays
   assign disp_blank = 1'b1;
   assign disp_clock = 1'b0;
   assign disp_rs = 1'b0;
   assign disp_ce_b = 1'b1;
   assign disp_reset_b = 1'b0;
   assign disp_data_out = 1'b0;

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

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

   // Daughtercard Connectors
   assign daughtercard = 44'hZ;

   // SystemACE Microprocessor Port
   assign systemace_data = 16'hZ;
   assign systemace_address = 7'h0;
   assign systemace_ce_b = 1'b1;
   assign systemace_we_b = 1'b1;
   assign systemace_oe_b = 1'b1;

   // Logic Analyzer
```

```
assign analyzer1_data = 16'h0;
assign analyzer1_clock = 1'b1;
assign analyzer2_data = 16'h0;
assign analyzer2_clock = 1'b1;
assign analyzer3_data = 16'h0;
assign analyzer3_clock = 1'b1;
assign analyzer4_data = 16'h0;
assign analyzer4_clock = 1'b1;

////////////////////////////////////////////////////////////////////////////
//
// Lab 4 Components
//
////////////////////////////////////////////////////////////////////////////


        wire clk;
        assign clk = clock_27mhz;


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

        wire button0_sync, button0_pulse;
        wire button1_sync, button1_pulse;
        wire button2_sync, button2_pulse;
        wire button3_sync, button3_pulse;
        wire button_up_sync, button_up_pulse;

        debounce db0 (reset, clk, ~button0, button0_sync);
        debounce db1 (reset, clk, ~button1, button1_sync);
        debounce db2 (reset, clk, ~button2, button2_sync);
        debounce db3 (reset, clk, ~button3, button3_sync);
        debounce db4 (reset, clk, ~button_up, button_up_sync);

        pulser puls0 (reset, clk, button0_sync, button0_pulse);
        pulser puls1 (reset, clk, button1_sync, button1_pulse);
        pulser puls2 (reset, clk, button2_sync, button2_pulse);
        pulser puls3 (reset, clk, button3_sync, button3_pulse);
        pulser puls4 (reset, clk, button_up_sync, button_up_pulse);

        /*
assign ram1_data = 36'hZ;
assign ram1_address = 19'h0;
assign ram1_adv_ld = 1'b0;
assign ram1_clk = 1'b0;
assign ram1_cen_b = 1'b1;
assign ram1_ce_b = 1'b1;
assign ram1_oe_b = 1'b1;
assign ram1_we_b = 1'b1;
assign ram1_bwe_b = 4'hF;
        */

        assign ram1_ce_b = 1'b0;
assign ram1_oe_b = 1'b0;
assign ram1_adv_ld = 1'b0;
```

```
   assign ram1_bwe_b = 4'h0;

   reg [3:0] A_songaddr;
   reg [11:0] A_noteaddr;
   reg A_start;

   wire A_done;
        wire A_exists;
   wire [35:0] A_note;
   wire [7:0] A_duration;

        // Port B: Writing to song memory.
        reg [3:0] B_songaddr;
        reg [11:0] B_noteaddr;
        reg [35:0] B_note;
        reg [7:0] B_duration;
        reg B_start;

        wire B_done;

        wire [3:0] C_song = 0;
        wire [3:0] D_song = 0;
        wire [3:0] E_song = 0;
        wire [3:0] F_song = 0;
        wire [3:0] G_song = 0;

        wire [255:0] C_name, D_name, E_name, F_name, G_name;
        wire C_start = 0;
        wire C_done;
        wire [3:0] Z_song = 0;
        wire [255:0] Z_name;
        wire Z_start = 0;
        wire Z_done;

   songmem sm1 (clk, reset,
                                ram1_clk, ram1_we_b, ram1_address, ram1_data,
ram1_cen_b,
                                A_songaddr, A_noteaddr, A_start, A_done, A_exists, A_note,
A_duration,
                                B_songaddr, B_noteaddr, B_note, B_duration, B_start,
B_done,
                                C_song, D_song, E_song, F_song, G_song,
                                C_name, D_name, E_name, F_name, G_name,
                                C_start, C_done,
                                Z_song, Z_name, Z_start, Z_done
                                );

        reg [7:0] led;
        reg wasup;

        always @ (posedge clk) begin
              A_start <= 0;
              B_start <= 0;
              if (reset) begin
                    wasup <= 0;
                    A_songaddr <= 0;
```

```
                        A_noteaddr <= 0;
                        B_songaddr <= 0;
                        B_noteaddr <= 0;
                        B_note <= 0;
                        B_duration <= 0;
                end else if (button0_pulse) begin // Read
                        wasup <= 0;
                        A_songaddr <= switch[7:4];
                        A_noteaddr <= {8'b0, switch[3:0]};
                        A_start <= 1;
                end else if (button_up_pulse) begin // Read
                        wasup <= 1;
                        A_songaddr <= switch[7:4];
                        A_noteaddr <= {8'b0, switch[3:0]};
                        A_start <= 1;
                end else if (button1_pulse) begin
                        B_songaddr <= switch[7:4];
                        B_noteaddr <= {8'b0, switch[3:0]};
                end else if (button2_pulse) begin
                        B_note <= {28'b0, switch[7:0]};
                end else if (button3_pulse) begin
                        B_duration <= switch[7:0];
                        B_start <= 1;
                end
        end


        always @ (posedge clk) begin
                if (reset) begin
                        led <= ~0;
                end else if (A_done && !wasup) begin
                        led <= ~A_note[7:0];
                end else if (A_done && wasup) begin
                        led <= ~A_duration[7:0];
                end else if (B_done) begin
                        led <= ~8'hFF;
                end
        end





//
// VGA output signals
//

// Inverting the clock to the DAC provides half a clock period for signals
// to propagate from the FPGA to the DAC.
assign vga_out_pixel_clock = 1'b1;

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

```
   // The following assignments should be deleted and replaced with your own
   // code to implement the Pong game.
   assign vga_out_red = 8'h0;
   assign vga_out_green = 8'h0;
   assign vga_out_blue = 8'h0;
   assign vga_out_blank_b = 1'b1;
   assign vga_out_hsync = 1'b0;
   assign vga_out_vsync = 1'b0;

endmodule





//------------ Songmem Module---------------------

// Interfaces with ZBT Memory to read and save songs.
// Simulates multiple ports for reading and writing, which can be activated simultaneously,
// but are serviced in sequence using an FSM.
//
// Handshaking pins indicate when to start a memory operation, and when the operation has completed.


module songmem(clk, reset,
						ram_clk, ram_we_b, ram_address, ram_data, ram_cen_b,
						A_songaddr, A_noteaddr, A_start, A_done, A_exists, A_note,
A_duration,
						B_songaddr, B_noteaddr, B_note, B_duration, B_start,
B_done,
						C_song, D_song, E_song, F_song, G_song,
						C_name, D_name, E_name, F_name, G_name,
						C_start, C_done,
						Z_song, Z_name, Z_start, Z_done
						);
   input clk;
	input reset;

   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


	// Port A: Reading from song memory.
   input [3:0] A_songaddr;
   input [11:0] A_noteaddr;
   input A_start;
   output A_done;
	output A_exists;
   output [35:0] A_note;
   output [7:0] A_duration;

	// Port B: Writing to song memory.
	input [3:0] B_songaddr;
input [11:0] B_noteaddr;
input [35:0] B_note;
input [7:0] B_duration;
input B_start;
output B_done;

// Ports C,D,E,F,G Reading names
input [3:0] C_song;  output [255:0] C_name;
input [3:0] D_song;           output [255:0] D_name;
input [3:0] E_song;           output [255:0] E_name;
input [3:0] F_song;  output [255:0] F_name;
input [3:0] G_song;  output [255:0] G_name;

input C_start;        // Begin reading names
output C_done;


reg [35:0] read_data; // Output after 4 cycles

// Ports for writing a name
input [3:0] Z_song;
input [255:0] Z_name;
input Z_start;
output Z_done;



// Latches for C,D,E,F,G
reg gotC;
reg [1279:0] C_giantbuffer;
reg [19:0] C_songlist;
reg [19:0] C_songlist_L;
reg [255:0] C_minibuffer;
reg [4:0] C_scount;


// C Output
assign C_name = C_giantbuffer[1279:1024];
assign D_name = C_giantbuffer[1023:768];
assign E_name = C_giantbuffer[767:512];
assign F_name = C_giantbuffer[511:256];
assign G_name = C_giantbuffer[255:0];
reg C_done;

// Z Latches
reg [3:0] Z_song_L;
reg [255:0] Z_name_L;
reg [255:0] Z_name_S;
// Z Output
reg Z_done;
reg gotZ;


// A Latches
reg gotA;
reg [3:0] A_songaddr_L;
reg [11:0] A_noteaddr_L;

// A Output
reg A_done;
reg A_exists;
reg [35:0] A_note;
reg [7:0] A_duration;


// B Latches
reg gotB;
reg [3:0] B_songaddr_L;
reg [11:0] B_noteaddr_L;
reg [35:0] B_note_L;
reg [7:0] B_duration_L;

// B Output
reg B_done;


always @ (posedge clk) begin
        if (reset) begin
                A_songaddr_L <= 0;
                A_noteaddr_L <= 0;
                gotA <= 0;
        end else if (!gotA && A_start) begin
                gotA <= 1;
                A_songaddr_L <= A_songaddr;
                A_noteaddr_L <= A_noteaddr;
        end else if (A_done) begin
                gotA <= 0;
        end
end

always @ (posedge clk) begin
        if (reset) begin
                gotB <= 0;
                B_songaddr_L <= 0;
                B_noteaddr_L <= 0;
                B_note_L <= 0;
                B_duration_L <= 0;
        end else if (!gotB && B_start) begin
                gotB <= 1;
                B_songaddr_L <= B_songaddr;
                B_noteaddr_L <= B_noteaddr;
                B_note_L <= B_note;
                B_duration_L <= B_duration;
        end else if (B_done) begin
                gotB <= 0;
        end
end



always @ (posedge clk) begin
```

```
        if (reset) begin
                gotC <= 0;
                C_songlist_L <= 0;
        end else if (!gotC && C_start) begin
                gotC <= 1;
                C_songlist_L <= {C_song, D_song, E_song, F_song, G_song};
        end else if (C_done) begin
                gotC <= 0;
        end
end



always @ (posedge clk) begin
        if (reset) begin
                gotZ <= 0;
                Z_song_L <= 0;
                Z_name_S <= 0;
        end else if (!gotZ && Z_start) begin
                gotZ <= 1;
                Z_song_L <= Z_song;
                Z_name_S <= Z_name;
        end else if (Z_done) begin
                gotZ <= 0;
        end
end





// Loop through all states, doing the action if necessary
reg [5:0] state;
parameter S_ZERO = 6'd0;
parameter S_CHECKA = 6'd1;
parameter S_CHECKB = 6'd2;
parameter S_CHECKC = 6'd3;
parameter S_CHECKZ = 6'd4;

parameter SA_PROC = 6'd5;
parameter SB_PROC = 6'd6;

parameter SC_PROC = 6'd7;
parameter SC_PROC_START = 6'd8;
parameter SC_PROC_SUB = 6'd9;
parameter SC_PROC_FIN = 6'd10;

parameter SZ_PROC = 6'd11;
parameter SZ_PROC_START = 6'd12;

reg [5:0] statecount;

reg weU;
reg [18:0] addrU;
reg [35:0] write_dataU;
```

```
always @ (posedge clk) begin
        if (reset) begin
                state <= S_ZERO;
                statecount <= 0;
                weU <= 0;
                addrU <= 0;
                A_done <= 0;
                B_done <= 0;
                C_scount <= 0;
                C_giantbuffer <= 0;
                C_minibuffer <= 0;
        end else begin
                statecount <= statecount + 1;
                case (state)
                S_ZERO: begin
                        weU <= 1;
                        addrU <= addrU+1;
                        write_dataU <= 36'b0;
                        if (addrU == 19'b1111111111111111111) begin
                                        state <= S_CHECKA;
                                        statecount <= 0;
                        end
                        end
                S_CHECKA: begin
                        if (gotA) state <= SA_PROC;
                        else state <= S_CHECKB;
                        statecount <= 0;
                        end
                S_CHECKB: begin
                        if (gotB) state <= SB_PROC;
                        else state <= S_CHECKC;
                        statecount <= 0;
                        end
                S_CHECKC: begin
                        if (gotC) state <= SC_PROC_START;
                        else state <= S_CHECKZ;
                        statecount <= 0;
                        end
                S_CHECKZ: begin
                        if (gotZ) state <= SZ_PROC_START;
                        else state <= S_CHECKA;
                        statecount <= 0;
                        end
                SA_PROC: begin
                        if (statecount == 0) begin
                                weU <= 0;
                                addrU <= {3'b0, A_songaddr_L, A_noteaddr_L};
                        end else if (statecount == 4) begin
                                A_note <= read_data[35:0];
                                addrU <= {3'b1, A_songaddr_L, A_noteaddr_L};
                        end else if (statecount == 8) begin
                                A_duration <= read_data[7:0];
                                A_exists <= read_data[35];
```

```
		end else if (statecount == 9) begin
			A_done <= 1;
		end else if (statecount == 10) begin
			A_done <= 0;
			state <= S_CHECKB;
			statecount <= 0;
		end
		end
	SB_PROC: begin
		if (statecount == 0) begin
			weU <= 1;
			addrU <= {3'b0, B_songaddr_L, B_noteaddr_L};
			write_dataU <= B_note_L;
		end else if (statecount == 1) begin
			weU <= 1;
			addrU <= {3'b1, B_songaddr_L, B_noteaddr_L};
			write_dataU <= {1'b1, 27'b0, B_duration_L};
		end else if (statecount == 2) begin
			B_done <= 1;
			weU <= 0;
		end else if (statecount == 3) begin
			B_done <= 0;
			state <= S_CHECKC;
			statecount <= 0;
		end
		end



	SC_PROC_START: begin
			C_songlist <= C_songlist_L;
			C_scount <= 0;
			C_giantbuffer <= 0;
			state <= SC_PROC_SUB;
			statecount <= 0;
		end

	SC_PROC: begin
		C_giantbuffer <= {C_minibuffer, C_giantbuffer[1279:256]};
		C_songlist <= {4'b0, C_songlist[15:0]};

		if (C_scount == 5) begin
			state <= SC_PROC_FIN;
			statecount <= 0;
		end else begin
			state <= SC_PROC_SUB;
			statecount <= 0;
		end
	end

	SC_PROC_FIN: begin
		if (statecount == 0) begin
			C_done <= 1;
		end else if (statecount == 1) begin
			C_done <= 0;
			state <= S_CHECKZ;
		end
	end

	SC_PROC_SUB: begin   // Read
		weU <= 0;
		if (statecount == 0) begin
			addrU <= {3'd2, C_songlist[3:0], 12'd0}; // First 32-bits
		end else if (statecount == 4) begin
			addrU <= {3'd2, C_songlist[3:0], 12'd1};
			C_minibuffer <= {C_minibuffer[255:32], read_data[31:0]};
		end else if (statecount == 8) begin
			addrU <= {3'd2, C_songlist[3:0], 12'd2};
			C_minibuffer <= {C_minibuffer[255:32], read_data[31:0]};
		end else if (statecount == 12) begin
			addrU <= {3'd2, C_songlist[3:0], 12'd3};
			C_minibuffer <= {C_minibuffer[255:32], read_data[31:0]};
		end else if (statecount == 16) begin
			addrU <= {3'd2, C_songlist[3:0], 12'd4};
			C_minibuffer <= {C_minibuffer[255:32], read_data[31:0]};
		end else if (statecount == 20) begin
			addrU <= {3'd2, C_songlist[3:0], 12'd5};
			C_minibuffer <= {C_minibuffer[255:32], read_data[31:0]};
		end else if (statecount == 24) begin
			addrU <= {3'd2, C_songlist[3:0], 12'd6};
			C_minibuffer <= {C_minibuffer[255:32], read_data[31:0]};
		end else if (statecount == 28) begin
			addrU <= {3'd2, C_songlist[3:0], 12'd7};
			C_minibuffer <= {C_minibuffer[255:32], read_data[31:0]};
		end else if (statecount == 32) begin
			C_minibuffer <= {C_minibuffer[255:32], read_data[31:0]};
		end else if (statecount == 36) begin
			state <= SC_PROC;
			statecount <= 0;
			C_scount <= C_scount+1;
		end
		end


	SZ_PROC_START:
		begin
		Z_name_L <= Z_name_S;
		state <= SZ_PROC;
		statecount <= 0;
		end

	SZ_PROC: begin

		if (statecount == 0) begin
			weU <= 1;
			addrU <= {3'd2, Z_song_L, 12'd0};
			write_dataU <= {4'b0, Z_name_L[255:224]};
			Z_name_L <= {Z_name_L[223:0], 32'b0};
		end else if (statecount == 1) begin
			weU <= 1;
			addrU <= {3'd2, Z_song_L, 12'd1};
			write_dataU <= {4'b0, Z_name_L[255:224]};
```

```
                        Z_name_L <= {Z_name_L[223:0], 32'b0};
                    end else if (statecount == 2) begin
                        weU <= 1;
                        addrU <= {3'd2, Z_song_L, 12'd2};
                        write_dataU <= {4'b0, Z_name_L[255:224]};
                        Z_name_L <= {Z_name_L[223:0], 32'b0};
                    end else if (statecount == 3) begin
                        weU <= 1;
                        addrU <= {3'd2, Z_song_L, 12'd3};
                        write_dataU <= {4'b0, Z_name_L[255:224]};
                        Z_name_L <= {Z_name_L[223:0], 32'b0};
                    end else if (statecount == 4) begin
                        weU <= 1;
                        addrU <= {3'd2, Z_song_L, 12'd4};
                        write_dataU <= {4'b0, Z_name_L[255:224]};
                        Z_name_L <= {Z_name_L[223:0], 32'b0};
                    end else if (statecount == 5) begin
                        weU <= 1;
                        addrU <= {3'd2, Z_song_L, 12'd5};
                        write_dataU <= {4'b0, Z_name_L[255:224]};
                        Z_name_L <= {Z_name_L[223:0], 32'b0};
                    end else if (statecount == 6) begin
                        weU <= 1;
                        addrU <= {3'd2, Z_song_L, 12'd6};
                        write_dataU <= {4'b0, Z_name_L[255:224]};
                        Z_name_L <= {Z_name_L[223:0], 32'b0};
                    end else if (statecount == 7) begin
                        weU <= 1;
                        addrU <= {3'd2, Z_song_L, 12'd7};
                        write_dataU <= {4'b0, Z_name_L[255:224]};
                        Z_name_L <= {Z_name_L[223:0], 32'b0};
                    end else if (statecount == 8) begin
                        weU <= 0;
                        Z_done <= 1;
                    end else if (statecount == 9) begin
                        Z_done <= 0;
                        state <= S_CHECKA;
                        statecount <= 0;
                    end
                    end

            endcase
        end
end



reg we;
reg [18:0] addr;
reg [35:0] write_data;


// Inputs to below: we, addr, write_data
// Outputs of below: read_data

always @ (posedge clk) begin
```

```
                        we <= weU;
                        addr <= addrU;
                        write_data <= write_dataU;
                end


   assign ram_clk = ~clk;
           assign ram_cen_b = 1'b0;

   reg [1:0]   we_delay;

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

   // create two-stage pipeline for write data

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

   assign      ram_we_b = ~we;
   assign      ram_address = addr;

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

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



endmodule


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

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

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

```
                count <= 0;
       end
     else if (count == DELAY)
       clean <= new;
     else
       count <= count+1;

endmodule
```

```
// Turns a long vsync into a one cycle pulse.

module pulser(reset, clock, in, out);
    input reset;
    input clock;
    input in;
    output out;

               reg out;
               reg pulsed;

               always @ (posedge clock)
                            begin
                                          if (reset) begin
                                                        out <= 0;
                                                        pulsed <= 0;
                                          end else if (pulsed == 0 && in == 1) begin
                                                        out <= 1;
                                                        pulsed <= 1;
                                          end else if (in == 0) begin
                                                        out <= 0;
                                                        pulsed <= 0;
                                          end else begin
                                                        out <= 0;
                                          end
                            end


endmodule
```

Step Interpretation

```
////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    10:40:49 05/16/06
// Design Name:
// Module Name:    beatselect
// Project Name:
```

```
// Target Device:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
module beatselect(clock, reset, beatflag, xcoord, ycoord, zcoord, nofeet, pressheight, nextdisplay, left,
right, enter, mode);


input clock, reset, beatflag;
input [31:0] xcoord, ycoord, zcoord;
input [14:0] pressheight;
input [1:0] mode;
input nofeet;

output [2:0] nextdisplay;
output left, right, enter;


reg [2:0] nextdisplay;
reg left, right, enter;

// Screen parameter numbers
parameter [2:0] main = 3'b000;
parameter [2:0] game = 3'b001;
parameter [2:0] beat = 3'b010;
parameter [2:0] record = 3'b011;
parameter [2:0] keyboard = 3'b100;

// Flags
reg feetup;          // Ensures button is not pressed because user's feet were in the right place when menu
appears
reg entered; // Ensures enter is zeroed after it is pressed



always @ (posedge clock)
begin
if((reset == 1) || (beatflag == 0))                                    // resets on logic high
        begin
        feetup <= 0;
        nextdisplay <= beat;
        left <= 0;
        right <= 0;
        enter <= 0;
        entered <= 0;
        end

else if(nofeet == 1)
        begin
```

```
		nextdisplay <= beat;
		left <= 0;
		right <= 0;
		enter <= 0;

		end

else											// If enter has been
stepped on, go to keyboard
		if(entered == 1)
			begin
			entered <= 0;
			enter <= 0;
			nextdisplay <= keyboard;
			end
		else if((zcoord <= pressheight) && (feetup == 1))	// After user has lifted feet, enable keys and
check
		begin
						// for button press
			if((xcoord >= 61) && (xcoord <= 155) && (ycoord >= 280) && (ycoord <= 406)) // left
button
				begin
				left <= 1;
				right <= 0;
				enter <= 0;
				nextdisplay <= beat;
				end
			else if((xcoord >= 201) && (xcoord <= 289) && (ycoord >= 280) && (ycoord <= 406))
// right button
				begin
				left <= 0;
				right <= 1;
				enter <= 0;
				nextdisplay <= beat;
				end
			else if((xcoord >= 342) && (xcoord <= 436) && (ycoord >= 280) && (ycoord <= 406))
// enter button
				begin
				left <= 0;
				right <= 0;
				enter <= 1;
				feetup <= 0;
				entered <= 1;
				end
			else if ((xcoord >= 483) && (xcoord <= 576) && (ycoord >= 280) && (ycoord <= 406))
// return button
				begin
				left <= 0;
				right <= 0;
				enter <= 0;
				feetup <= 0;
				nextdisplay <= main;
				end
			end
else if((zcoord >= (pressheight + 100)) &&		// Do not enable buttons until user lifts feet
                                    (((xcoord >= 61) && (xcoord <= 155) && (ycoord >= 280) && (ycoord <=
406)) ||
                                     ((xcoord >= 201) && (xcoord <= 289) && (ycoord >= 280) && (ycoord <=
406)) ||
                                     ((xcoord >= 342) && (xcoord <= 436) && (ycoord >= 280) && (ycoord <=
406))||
                                     ((xcoord >= 483) && (xcoord <= 576) && (ycoord >= 280) && (ycoord <=
406))))
                                     begin
                                     feetup <= 1;
                                     //nextdisplay <= beat;
                                     end
//else nextdisplay <= beat;

end


endmodule


module boardsel(clock, reset, screennumout, xcoord1, ycoord1, zcoord1, nofoot1,
                         xcoord2, ycoord2, zcoord2, nofoot2, pressheight, up, down, left, right, delete, enter,
select,
                         mode, keynum, keypress);

input clock, reset;

input [31:0] xcoord1, ycoord1, zcoord1, xcoord2, ycoord2, zcoord2;
input [14:0] pressheight;

input nofoot1, nofoot2;


output [35:0] keynum; // For Audio and Projection
output keypress;  // If key is being pressed -- for Audio



wire [35:0] keynum1, keynum2; // For Audio and Projection
wire keypress1, keypress2;             // If key is being pressed -- for Audio
reg [35:0] keynum;
reg keypress;


output [2:0] screennumout;

reg [2:0] screennumout;


wire [2:0] nextdisplay0, nextdisplay1, nextdisplay2, nextdisplay3, nextdisplay4;
wire [2:0] nextdisplay00, nextdisplay11, nextdisplay22, nextdisplay33, nextdisplay44;


////////////////////// To Audio

output [1:0] mode;
```

```
wire [1:0] mode1, mode2;
reg mode;

/////////////////////// To projection
output up, down, left, right, delete, enter, select;

reg up, down, left, right, delete, enter, select;
wire upg, upr, downg, downr, leftb, rightb, leftr, rightr, deleter, enterr, selectr, enterg, enterb;
wire upgg, uprr, downgg, downrr, leftbb, rightbb, leftrr, rightrr, deleterr, enterrr, selectrr, entergg, enterbb;


/////////////////////////Menu Flags -- The module controling each menu only changes output based
// on user input if the particular menu's flag is logic high. Other menus do not change output
// when one menu is being projected ////////////////
reg mainflag, keyflag, gameflag, recordflag, beatflag;



// Display menu numbers
parameter [2:0] main = 3'b000;
parameter [2:0] game = 3'b001;
parameter [2:0] beat = 3'b010;
parameter [2:0] record = 3'b011;
parameter [2:0] keyboard = 3'b100;


// Instantiation of key step detection          --- FOOT 1 ------
mainmenu menured(clock, reset, mainflag, xcoord1, ycoord1, zcoord1, nofoot1, pressheight, nextdisplay0,
mode1);
gamemenu gamered(clock, reset, gameflag, xcoord1, ycoord1, zcoord1, nofoot1, pressheight, nextdisplay1,
upg, downg, enterg);
recordname recordred(clock, reset, recordflag, xcoord1, ycoord1, zcoord1, nofoot1, pressheight,
nextdisplay2, upr, downr, leftr, rightr, deleter, enterr, selectr);
beatselect beatred(clock, reset, beatflag, xcoord1, ycoord1, zcoord1, nofoot1, pressheight, nextdisplay3,
leftb, rightb, enterb, mode);
BoardDetect boardred(reset, clock, keyflag, xcoord1, ycoord1, zcoord1, nofoot1, keynum1, keypress1,
pressheight, nextdisplay4);

// Instantiation of key step detection          --- FOOT 2 ------
mainmenu menublue(clock, reset, mainflag, xcoord2, ycoord2, zcoord2, nofoot2, pressheight,
nextdisplay00, mode2);
gamemenu gameblue(clock, reset, gameflag, xcoord2, ycoord2, zcoord2, nofoot2, pressheight,
nextdisplay11, upgg, downgg, entergg);
recordname recordblue(clock, reset, recordflag, xcoord2, ycoord2, zcoord2, nofoot2, pressheight,
nextdisplay22, uprr, downrr, leftrr, rightrr, deleterr, enterrr, selectrr);
beatselect beatblue(clock, reset, beatflag, xcoord2, ycoord2, zcoord2, nofoot2, pressheight, nextdisplay33,
leftbb, rightbb, enterbb, mode);
BoardDetect boardblue(reset, clock, keyflag, xcoord2, ycoord2, zcoord2, nofoot2, keynum2, keypress2,
pressheight, nextdisplay44);


always @ (posedge clock)

begin
if(reset == 1)                                        // resets on logic high - all output zeroed
        begin
```

```
screennumout <= main;
up <= 0;
down <= 0;
left <= 0;
right <= 0;
enter <= 0;
delete <= 0;
select <= 0;
mainflag <= 0;
keyflag <= 0;
gameflag <= 0;
recordflag <= 0;
beatflag <= 0;

end
else
        case (screennumout)      // Determines which screen the user currently sees projected
                main:
 // MAIN MENU
                        begin
         // Use output of main menu module

                                        mainflag <= 1;
                                        keyflag <= 0;
                                        gameflag <= 0;
                                        recordflag <= 0;
                                        beatflag <= 0;

                                up <= 0;
                                down <= 0;
                                left <= 0;
                                right <= 0;
                                enter <= 0;
                                delete <= 0;
                                select <= 0;

                                keynum <= 0;
                                keypress <= 0;

                                mode <= (mode1 | mode2);


                                if(nextdisplay0 != main)
                                        screennumout <= nextdisplay0;
                                else if(nextdisplay00 != main)
                                        screennumout <= nextdisplay00;
                                else    screennumout <= nextdisplay0;



                        end
                game:
        // GAME MENU
                        begin
                // Use output of game menu module
                                up <= (upg | upgg);
```

```
                    down <= (downg | downgg);
                    enter <= (enterg | entergg);

                left <= 0;
                right <= 0;
                delete <= 0;
                select <= 0;


                keynum <= 0;
                keypress <= 0;

                        mainflag <= 0;
                        keyflag <= 0;
                        gameflag <= 1;
                        recordflag <= 0;
                        beatflag <= 0;

                    if(nextdisplay1 != game)
                        screennumout <= nextdisplay1;
                    else if(nextdisplay11 != game)
                        screennumout <= nextdisplay11;
                    else    screennumout <= nextdisplay1;


                end
            record:
// RECORD MENU
                begin
        // Use output of record menu module
                    up <= (upr | uprr);
                    down <= (downr | downrr);
                    left <= (leftr | leftrr);
                    right <= (rightr | rightrr);
                    enter <= (enterr | enterrr);
                    delete <= (deleter | deleterr);
                    select <= (selectr | selectrr);

                    keynum <= 0;
                    keypress <= 0;


                        mainflag <= 0;
                        keyflag <= 0;
                        gameflag <= 0;
                        recordflag <= 1;
                        beatflag <= 0;

                    if(nextdisplay2 != record)
                        screennumout <= nextdisplay2;
                    else if(nextdisplay22 != record)
                        screennumout <= nextdisplay22;
                    else    screennumout <= nextdisplay2;


                end
```

```
                        beat:
// BEAT SELECTION MENU
                                begin
        // Use output of beat selection menu module
                                        left <= (leftb | leftbb);
                                        right <= (rightb | rightbb);
                                        enter <= (enterb | enterbb);

                                up <= 0;
                                down <= 0;
                                delete <= 0;
                                select <= 0;
                                keypress <= 0;
                                keynum <= 0;

                                                mainflag <= 0;
                                                keyflag <= 0;
                                                gameflag <= 0;
                                                recordflag <= 0;
                                                beatflag <= 1;

                                        if(nextdisplay3 != beat)
                                                screennumout <= nextdisplay3;
                                        else if(nextdisplay33 != beat)
                                                screennumout <= nextdisplay33;
                                        else    screennumout <= nextdisplay3;


                                end
                        keyboard:
       // The Keyboard is beign displayed
                                begin

                                up <= 0;
                                down <= 0;
                                left <= 0;
                                right <= 0;
                                enter <= 0;
                                delete <= 0;
                                select <= 0;


                                                mainflag <= 0;
                                                keyflag <= 1;
                                                gameflag <= 0;
                                                recordflag <= 0;
                                                beatflag <= 0;

                                        keypress <= (keypress1 | keypress2);
                                        keynum <= (keynum1 | keynum2);


                                        if(nextdisplay4 != keyboard)
                                                screennumout <= nextdisplay4;
                                        else if(nextdisplay44 != keyboard)
                                                screennumout <= nextdisplay44;
                                        else    screennumout <= nextdisplay4;


                                end
                        default:
        // Unrecognizable display defaults to main menu
                                begin
                                        screennumout <= main;

                                                mainflag <= 0;
                                                keyflag <= 0;
                                                gameflag <= 0;
                                                recordflag <= 0;
                                                beatflag <= 0;

                                end
                endcase
end


endmodule







////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    10:38:31 05/16/06
// Design Name:
// Module Name:    gamemenu
// Project Name:
// Target Device:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
module gamemenu(clock, reset, gameflag, xcoord, ycoord, zcoord, nofeet, pressheight, nextdisplay, up,
down, enter);

input clock, reset, gameflag;
input [31:0] xcoord, ycoord, zcoord;
input [14:0] pressheight;
input nofeet;
```

```
output [2:0] nextdisplay;
output up, down, enter;


reg [2:0] nextdisplay;
reg up, down, enter;

// Screen number parameters
parameter [2:0] main = 3'b000;
parameter [2:0] game = 3'b001;
parameter [2:0] beat = 3'b010;
parameter [2:0] record = 3'b011;
parameter [2:0] keyboard = 3'b100;

// Flags
reg feetup;
reg entered;



always @ (posedge clock)
begin
if((reset == 1) || (gameflag == 0))                              // resets on logic high
        begin
        feetup <= 0;
        nextdisplay <= game;
        up <= 0;
        down <= 0;
        enter <= 0;
        entered <= 0;
        end
else if(nofeet == 1)
        begin

        nextdisplay <= game;
        up <= 0;
        down <= 0;
        enter <= 0;

        end




else
        if(entered == 1)    // If enter button is pressed, switch to Beat Selection Menu
                begin
                entered <= 0;
                enter <= 0;
                nextdisplay <= beat;
                end

else
        if((zcoord <= pressheight) && (feetup == 1))          //Once buttons are enabled, check if button
is pressed
        begin
```

```
                if((xcoord >= 61) && (xcoord <= 155) && (ycoord >= 280) && (ycoord <= 406))  // up
button
                        begin
                        up <= 1;
                        down <= 0;
                        enter <= 0;
                        nextdisplay <= game;
                        end
                else if((xcoord >= 201) && (xcoord <= 289) && (ycoord >= 280) && (ycoord <= 406))
// down button
                        begin
                        up <= 0;
                        down <= 1;
                        enter <= 0;
                        nextdisplay <= game;
                        end
                else if((xcoord >= 342) && (xcoord <= 436) && (ycoord >= 280) && (ycoord <= 406))
// enter button
                        begin
                        up <= 0;
                        down <= 0;
                        enter <= 1;
                        feetup <= 0;
                        entered <= 1;
                        end
                else if((xcoord >= 483) && (xcoord <= 576) && (ycoord >= 280) && (ycoord <= 406))
// return button
                        begin
                        up <= 0;
                        down <= 0;
                        enter <= 0;
                        feetup <= 0;
                        nextdisplay <= main;
                        end
                end

else if((zcoord >= (pressheight + 100)) &&     // Button enabling process (feetup = 0)
                        (((xcoord >= 61) && (xcoord <= 155) && (ycoord >= 280) && (ycoord <=
406)) ||
                         ((xcoord >= 201) && (xcoord <= 289) && (ycoord >= 280) && (ycoord <=
406)) ||
                         ((xcoord >= 342) && (xcoord <= 436) && (ycoord >= 280) && (ycoord <=
406))||
                         ((xcoord >= 483) && (xcoord <= 576) && (ycoord >= 280) && (ycoord <=
406))))
                         begin
                         feetup <= 1;
                         //nextdisplay <= game;
                         end
//else nextdisplay <= game;

end

endmodule
```

```
////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    10:37:48 05/16/06
// Design Name:
// Module Name:    mainmenu
// Project Name:
// Target Device:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
module mainmenu(clock, reset, mainflag, xcoord, ycoord, zcoord, nofeet, pressheight, nextdisplay, mode);

input clock, reset, mainflag;
input [31:0] xcoord, ycoord, zcoord;
input [14:0] pressheight;
input nofeet;


output [2:0] nextdisplay;
output [1:0] mode;

reg [2:0] nextdisplay;
reg [1:0] mode;




// MENU SELECT
parameter [2:0] main = 3'b000;
parameter [2:0] game = 3'b001;
parameter [2:0] beat = 3'b010;
parameter [2:0] record = 3'b011;
parameter [2:0] keyboard = 3'b100;


// MODE SELECT
parameter [1:0] nomode = 2'b00;
parameter [1:0] playmode = 2'b01;
parameter [1:0] recordmode = 2'b10;
parameter [1:0] gamemode = 2'b11;
```

```
// Flag ---- the user must lift his feet up and step down again to be able to select a button
// This guards against a menu changing because a user's feet happened to be in the right place
// when the menu appears
reg feetup;


always @ (posedge clock)
begin
        if((reset == 1) || (mainflag == 0))                    // Reset on logic high
                begin
                feetup <= 0;
                nextdisplay <= main;
                mode <= nomode;
                end


else if(nofeet == 1)
                begin

                nextdisplay <= main;
                mode <= nomode;
                end


                else
                        if((zcoord <= pressheight) && (feetup == 1))            // After enabling buttons,
check if any are pressed

                                        if((xcoord >= 60) && (xcoord <= 300) && (ycoord >= 80)
&& (ycoord <= 220))     // Play mode
                                                begin
                                                nextdisplay <= keyboard;
                                                mode <= playmode;
                                                feetup <= 0;
                                                end
                                        else if((xcoord >= 340) && (xcoord <= 580) && (ycoord >=
80) && (ycoord <= 220)) // Game mode
                                                begin
                                                nextdisplay <= game;
                                                mode <= gamemode;
                                                feetup <= 0;
                                                end
                                        else if((xcoord >= 200) && (xcoord <= 440) && (ycoord >=
280) && (ycoord <= 420))  // Record mode
                                                begin
                                                nextdisplay <= record;
                                                mode <= recordmode;
                                                feetup <= 0;
                                                end
                                        else
        // If no modes are stepped on, the display stays on Main Menu
                                                begin
                                                nextdisplay <= main;
                                                mode <= nomode;
                                                end
```

```
				else if((zcoord >= (pressheight + 100)) &&	// Button enabling process
										(((xcoord >= 60) && (xcoord <=
300) && (ycoord >= 80) && (ycoord <= 220)) ||
										((xcoord >= 340) && (xcoord <=
580) && (ycoord >= 80) && (ycoord <= 220)) ||
										((xcoord >= 200) && (xcoord <=
440) && (ycoord >= 280) && (ycoord <= 420))))
								begin
								feetup <= 1;
								//nextdisplay <= main;
								end
				/*		else begin
							nextdisplay <= main;
							end*/

end





endmodule





////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    10:39:08 05/16/06
// Design Name:
// Module Name:    recordname
// Project Name:
// Target Device:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
module recordname(clock, reset, recordflag, xcoord, ycoord, zcoord, nofeet, pressheight, nextdisplay, up,
down, left, right, delete, enter, select);


input clock, reset, recordflag;
input [31:0] xcoord, ycoord, zcoord;
input [14:0] pressheight;
input nofeet;


output [2:0] nextdisplay;
output up, down, left, right, delete, enter, select;


reg [2:0] nextdisplay;
reg up, down, left, right, delete, enter, select;

// Screen number parameters
parameter [2:0] main = 3'b000;
parameter [2:0] game = 3'b001;
parameter [2:0] beat = 3'b010;
parameter [2:0] record = 3'b011;
parameter [2:0] keyboard = 3'b100;

// Flags
reg feetup;
reg entered;



always @ (posedge clock)
begin
if((reset == 1) || (recordflag == 0))                           // resets on logic high
        begin
        feetup <= 0;
        nextdisplay <= record;
        up <= 0;
        down <= 0;
        left <= 0;
        right <= 0;
        delete <= 0;
        enter <= 0;
        select <= 0;
        entered <= 0;
        end

else if(nofeet == 1)
        begin

        nextdisplay <= record;
        up <= 0;
        down <= 0;
        left <= 0;
        right <= 0;
        delete <= 0;
        enter <= 0;
        select <= 0;

        end
else
        if(entered == 1)            // If enter button is pressed, switch to Beat Selection Menu
                begin
                entered <= 0;
```

```
				enter <= 0;
				nextdisplay <= beat;
				end
else
		if((zcoord <= pressheight) && (feetup == 1))		//Once buttons are enabled, check if any
are pressed
		begin
				if((xcoord >= 61) && (xcoord <= 155) && (ycoord >= 369) && (ycoord <= 469))  // up
button
						begin
						up <= 1;
						down <= 0;
						left <= 0;
						right <= 0;
						delete <= 0;
						enter <= 0;
						select <= 0;
						nextdisplay <= record;
						end
				else if((xcoord >= 201) && (xcoord <= 289) && (ycoord >= 369) && (ycoord <= 469))
// down button
						begin
						up <= 0;
						down <= 1;
						left <= 0;
						right <= 0;
						delete <= 0;
						enter <= 0;
						select <= 0;
						nextdisplay <= record;
						end
				else if((xcoord >= 342) && (xcoord <= 436) && (ycoord >= 369) && (ycoord <= 469))
// left button
						begin
						up <= 0;
						down <= 0;
						left <= 1;
						right <= 0;
						delete <= 0;
						enter <= 0;
						select <= 0;
						nextdisplay <= record;
						end
				else if((xcoord >= 483) && (xcoord <= 576) && (ycoord >= 369) && (ycoord <= 469))
// right button
						begin
						up <= 0;
						down <= 0;
						left <= 0;
						right <= 1;
						delete <= 0;
						enter <= 0;
						select <= 0;
						nextdisplay <= record;
						end
```

```
                else if((xcoord >= 342) && (xcoord <= 436) && (ycoord >= 249) && (ycoord <= 349))
// delete button
                        begin
                        up <= 0;
                        down <= 0;
                        left <= 0;
                        right <= 0;
                        delete <= 1;
                        enter <= 0;
                        select <= 0;
                        nextdisplay <= record;
                        end
                else if((xcoord >= 61) && (xcoord <= 155) && (ycoord >= 249) && (ycoord <= 349))
// select button
                        begin
                        up <= 0;
                        down <= 0;
                        left <= 0;
                        right <= 0;
                        delete <= 0;
                        enter <= 0;
                        select <= 1;
                        nextdisplay <= record;
                        end
                else if((xcoord >= 201) && (xcoord <= 289) && (ycoord >= 249) && (ycoord <= 349))
// enter button
                        begin
                        up <= 0;
                        down <= 0;
                        left <= 0;
                        right <= 0;
                        delete <= 0;
                        enter <= 1;
                        select <= 0;
                        entered <= 1;
                        feetup <= 0;
                        end
                else if((xcoord >= 483) && (xcoord <= 576) && (ycoord >= 249) && (ycoord <= 349))
// return button
                        begin
                        up <= 0;
                        down <= 0;
                        left <= 0;
                        right <= 0;
                        delete <= 0;
                        enter <= 0;
                        select <= 0;
                        nextdisplay <= main;
                        feetup <= 0;
                        end
                end
else if((zcoord >= (pressheight + 100)) &&                 // Button enabling process
                        (((xcoord >= 61) && (xcoord <= 155) && (ycoord >= 369) && (ycoord <=
469)) ||                 //up
                         ((xcoord >= 201) && (xcoord <= 289) && (ycoord >= 369) && (ycoord <=
469)) ||          //down
```

```
                                ((xcoord >= 342) && (xcoord <= 436) && (ycoord >= 369) && (ycoord <=
469)) ||                //left
                                ((xcoord >= 483) && (xcoord <= 576) && (ycoord >= 369) && (ycoord <=
469)) ||                //right
                                ((xcoord >= 342) && (xcoord <= 436) && (ycoord >= 249) && (ycoord <=
349)) ||                //delete
                                ((xcoord >= 61) && (xcoord <= 155) && (ycoord >= 249) && (ycoord <=
349)) ||                        // select
                                ((xcoord >= 201) && (xcoord <= 289) && (ycoord >= 249) && (ycoord <=
349)) ||                //enter
                                ((xcoord >= 483) && (xcoord <= 576) && (ycoord >= 249) && (ycoord <=
349))))                         //return
                                begin
                                feetup <= 1;
                                //nextdisplay <= record;
                                end
//else nextdisplay <= record;

end

endmodule




////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    10:39:55 05/16/06
// Design Name:
// Module Name:    BoardDetect
// Project Name:
// Target Device:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
module BoardDetect(reset, clock, keyflag, xcoord, ycoord, zcoord, nofeet, keynum, keypress, pressheight,
nextdisplay);


input reset, clock, keyflag;
input [31:0] xcoord, ycoord, zcoord;
input [14:0] pressheight;
input nofeet;

output [35:0] keynum; // For Audio and Projection
output keypress;   // If key is being pressed -- for Audio
output [2:0] nextdisplay;

reg [35:0] keynum;
reg keypress;
reg [2:0] nextdisplay;


reg feetup;


// Menu Options
parameter [2:0] main = 3'b000;
parameter [2:0] game = 3'b001;
parameter [2:0] beat = 3'b010;
parameter [2:0] record = 3'b011;
parameter [2:0] keyboard = 3'b100;


// Key number
parameter [35:0] nopress =          36'b000000000000000000000000000000000000;
parameter [35:0] zero =             36'b000000000000000000000000000000000001;
parameter [35:0] one =                      36'b000000000000000000000000000000000010;
parameter [35:0] two =                      36'b000000000000000000000000000000000100;
parameter [35:0] three =            36'b000000000000000000000000000000001000;
parameter [35:0] four =             36'b000000000000000000000000000000010000;
parameter [35:0] five =             36'b000000000000000000000000000000100000;
parameter [35:0] six =                      36'b000000000000000000000000000001000000;
parameter [35:0] seven =            36'b000000000000000000000000000010000000;
parameter [35:0] eight =            36'b000000000000000000000000000100000000;
parameter [35:0] nine =             36'b000000000000000000000000001000000000;
parameter [35:0] ten =                      36'b000000000000000000000000010000000000;
parameter [35:0] eleven =           36'b000000000000000000000000100000000000;
parameter [35:0] twelve =           36'b000000000000000000000001000000000000;
parameter [35:0] thirteen =         36'b000000000000000000000010000000000000;
parameter [35:0] fourteen =         36'b000000000000000000000100000000000000;
parameter [35:0] fifteen =  36'b000000000000000000001000000000000000;
parameter [35:0] sixteen =          36'b000000000000000000010000000000000000;
parameter [35:0] seventeen =        36'b000000000000000000100000000000000000;
parameter [35:0] eighteen =         36'b000000000000000001000000000000000000;
parameter [35:0] nineteen =         36'b000000000000000010000000000000000000;
parameter [35:0] twenty =           36'b000000000000000100000000000000000000;
parameter [35:0] twenone =          36'b000000000000001000000000000000000000;
parameter [35:0] twentwo =          36'b000000000000010000000000000000000000;
parameter [35:0] twenthree =        36'b000000000000100000000000000000000000;
parameter [35:0] twenfour =         36'b000000000001000000000000000000000000;
parameter [35:0] twenfive =         36'b000000000010000000000000000000000000;
parameter [35:0] twensix =          36'b000000000100000000000000000000000000;
parameter [35:0] twenseven =        36'b000000001000000000000000000000000000;
parameter [35:0] tweneight =        36'b000000010000000000000000000000000000;
parameter [35:0] twennine =         36'b000000100000000000000000000000000000;
parameter [35:0] thirty =           36'b000001000000000000000000000000000000;
parameter [35:0] thirone =          36'b000010000000000000000000000000000000;
parameter [35:0] thirtwo =          36'b000100000000000000000000000000000000;
parameter [35:0] thirthree =        36'b001000000000000000000000000000000000;
parameter [35:0] thirfour =         36'b010000000000000000000000000000000000;
parameter [35:0] thirfive =              36'b100000000000000000000000000000000000;


// Key size parameters
parameter blackw = 58;
parameter blackh = 83;
parameter whitew = 75;
parameter whiteh = 160;




always @ (posedge clock)
begin
if((reset == 1) || (keyflag == 0))                                                    // reset on logic high
            begin
            keynum <= nopress;
            keypress <= 0;
            nextdisplay <= keyboard;
            feetup <= 0;
            end

else if(nofeet == 1)
            begin
            keynum <= nopress;
            keypress <= 0;
            nextdisplay <= keyboard;

            end



else
 if((zcoord >= (pressheight + 100)) && (xcoord >= 564) && (xcoord <= 639))    // Check if user has lifted
feet
                        begin
                                                        // "Return" button requires lifting feet enable
                        keypress <= 0;
                                      // Key press to play notes does not require enable
                        keynum <= nopress;
                        //nextdisplay <= keyboard;
                        feetup <= 1;
                        end
else if(zcoord >= pressheight)                                // If feet are lifted, no keys are pressed
                        begin
                        keypress <= 0;
                        keynum <= nopress;
                        //nextdisplay <= keyboard;
                        end
            else
              begin

                        if((xcoord >= 564) && (xcoord <= 639) && (feetup == 1))  //After enabling Return, if
Return
                                    begin
                                                                                    //is stepped on, return to main
menu
```

```
					nextdisplay <= main;
					feetup <= 0;
					keypress <= 0;
					keynum <= nopress;
					end
				else
//--------- BLACK KEYS --------------		Check if/which black keys are stepped on

				begin


				if((xcoord >= 44) && (xcoord <= (44 + blackw)) && (ycoord >= 0) && (ycoord <=
blackh))
					begin
					keypress <= 1;
					keynum <= twenone;
					end
				else if((xcoord >= 118) && (xcoord <= (118 + blackw)) && (ycoord >= 0) && (ycoord
<= blackh))
					begin
					keypress <= 1;
					keynum <= twentwo;
					end
				else if((xcoord >= 268) && (xcoord <= (268 + blackw)) && (ycoord >= 0) && (ycoord
<= blackh))
					begin
					keypress <= 1;
					keynum <= twenthree;
					end
				else if((xcoord >= 343) && (xcoord <= (343 + blackw)) && (ycoord >= 0) && (ycoord
<= blackh))
					begin
					keypress <= 1;
					keynum <= twenfour;
					end
				else if((xcoord >= 418) && (xcoord <= (418 + blackw)) && (ycoord >= 0) && (ycoord
<= blackh))
					begin
					keypress <= 1;
					keynum <= twenfive;
					end
				else if((xcoord >= 44) && (xcoord <= (44 + blackw)) && (ycoord >= 160) && (ycoord
<= (160 + blackh)))
					begin
					keypress <= 1;
					keynum <= twensix;
					end
				else if((xcoord >= 118) && (xcoord <= (118 + blackw)) && (ycoord >= 160) &&
(ycoord <= (160 + blackh)))
					begin
					keypress <= 1;
					keynum <= twenseven;
					end
				else if((xcoord >= 268) && (xcoord <= (268 + blackw)) && (ycoord >= 160) &&
(ycoord <= (160 + blackh)))
					begin
				keypress <= 1;
				keynum <= tweneight;
				end
			else if((xcoord >= 343) && (xcoord <= (343 + blackw)) && (ycoord >= 160) &&
(ycoord <= (160 + blackh)))
				begin
				keypress <= 1;
				keynum <= twennine;
				end
			else if((xcoord >= 418) && (xcoord <= (418 + blackw)) && (ycoord >= 160) &&
(ycoord <= (160 + blackh)))
				begin
				keypress <= 1;
				keynum <= thirty;
				end
			else if((xcoord >= 44) && (xcoord <= (44 + blackw)) && (ycoord >= 320) && (ycoord
<= (320 + blackh)))
				begin
				keypress <= 1;
				keynum <= thirone;
				end
			else if((xcoord >= 118) && (xcoord <= (118 + blackw)) && (ycoord >= 320) &&
(ycoord <= (320 + blackh)))
				begin
				keypress <= 1;
				keynum <= thirtwo;
				end
			else if((xcoord >= 268) && (xcoord <= (268 + blackw)) && (ycoord >= 320) &&
(ycoord <= (320 + blackh)))
				begin
				keypress <= 1;
				keynum <= thirthree;
				end
			else if((xcoord >= 343) && (xcoord <= (343 + blackw)) && (ycoord >= 320) &&
(ycoord <= (320 + blackh)))
				begin
				keypress <= 1;
				keynum <= thirfour;
				end
			else if((xcoord >= 418) && (xcoord <= (504 + blackw)) && (ycoord >= 320) &&
(ycoord <= (320 + blackh)))
				begin
				keypress <= 1;
				keynum <= thirfive;
				end

			// --------- WHITE KEYS ---------    Check if/which white keys are being stepped on
			else if((xcoord >= 1) && (xcoord <= (1 + whitew)) && (ycoord >= 0) && (ycoord <= (0
+ whiteh)))
				begin
				keypress <= 1;
				keynum <= zero;
				end
			else if((xcoord >= 75) && (xcoord <= (75 + whitew)) && (ycoord >= 0) && (ycoord <=
(0 + whiteh)))
				begin
```

```
						keypress <= 1;
						keynum <= one;
						end
				else if((xcoord >= 150) && (xcoord <= (150 + whitew)) && (ycoord >= 0) && (ycoord
<= (0 + whiteh)))
						begin
						keypress <= 1;
						keynum <= two;
						end
				else if((xcoord >= 226) && (xcoord <= (226 + whitew)) && (ycoord >= 0) && (ycoord
<= (0 + whiteh)))
						begin
						keypress <= 1;
						keynum <= three;
						end
				else if((xcoord >= 301) && (xcoord <= (301 + whitew)) && (ycoord >= 0) && (ycoord
<= (0 + whiteh)))
						begin
						keypress <= 1;
						keynum <= four;
						end
				else if((xcoord >= 376) && (xcoord <= (376 + whitew)) && (ycoord >= 0) && (ycoord
<= (0 + whiteh)))
						begin
						keypress <= 1;
						keynum <= five;
						end
				else if((xcoord >= 451) && (xcoord <= (451 + whitew)) && (ycoord >= 0) && (ycoord
<= (0 + whiteh)))
						begin
						keypress <= 1;
						keynum <= six;
						end
				else if((xcoord >= 1) && (xcoord <= (1 + whitew)) && (ycoord >= 160) && (ycoord <=
(160 + whiteh)))
						begin
						keypress <= 1;
						keynum <= seven;
						end
				else if((xcoord >= 75) && (xcoord <= (75 + whitew)) && (ycoord >= 160) && (ycoord
<= (160 + whiteh)))
						begin
						keypress <= 1;
						keynum <= eight;
						end
				else if((xcoord >= 150) && (xcoord <= (150 + whitew)) && (ycoord >= 160) &&
(ycoord <= (160 + whiteh)))
						begin
						keypress <= 1;
						keynum <= nine;
						end
				else if((xcoord >= 226) && (xcoord <= (226 + whitew)) && (ycoord >= 160) &&
(ycoord <= (160 + whiteh)))
						begin
						keypress <= 1;
						keynum <= ten;
```

```
                        end
                else if((xcoord >= 301) && (xcoord <= (301 + whitew)) && (ycoord >= 160) &&
(ycoord <= (160 + whiteh)))
                        begin
                        keypress <= 1;
                        keynum <= eleven;
                        end
                else if((xcoord >= 376) && (xcoord <= (376 + whitew)) && (ycoord >= 160) &&
(ycoord <= (160 + whiteh)))
                        begin
                        keypress <= 1;
                        keynum <= twelve;
                        end
                else if((xcoord >= 451) && (xcoord <= (451 + whitew)) && (ycoord >= 160) &&
(ycoord <= (160 + whiteh)))
                        begin
                        keypress <= 1;
                        keynum <= thirteen;
                        end
                else if((xcoord >= 1) && (xcoord <= (1 + whitew)) && (ycoord >= 320) && (ycoord <=
(320 + whiteh)))
                        begin
                        keypress <= 1;
                        keynum <= fourteen;
                        end
                else if((xcoord >= 75) && (xcoord <= (75 + whitew)) && (ycoord >= 320) && (ycoord
<= (320 + whiteh)))
                        begin
                        keypress <= 1;
                        keynum <= fifteen;
                        end
                else if((xcoord >= 150) && (xcoord <= (150 + whitew)) && (ycoord >= 320) &&
(ycoord <= (320 + whiteh)))
                        begin
                        keypress <= 1;
                        keynum <= sixteen;
                        end
                else if((xcoord >= 226) && (xcoord <= (226 + whitew)) && (ycoord >= 320) &&
(ycoord <= (320 + whiteh)))
                        begin
                        keypress <= 1;
                        keynum <= seventeen;
                        end
                else if((xcoord >= 301) && (xcoord <= (301 + whitew)) && (ycoord >= 320) &&
(ycoord <= (320 + whiteh)))
                        begin
                        keypress <= 1;
                        keynum <= eighteen;
                        end
                else if((xcoord >= 376) && (xcoord <= (376 + whitew)) && (ycoord >= 320) &&
(ycoord <= (320 + whiteh)))
                        begin
                        keypress <= 1;
                        keynum <= nineteen;
                        end
```

```
                    else if((xcoord >= 451) && (xcoord <= (451 + whitew)) && (ycoord >= 320) &&
(ycoord <= (320 + whiteh)))
                              begin
                              keypress <= 1;
                              keynum <= twenty;
                              end
                    else
                              begin
                              keynum <= nopress;
                              keypress <= 0;
                              end
                    end
          end


end




endmodule
```

Video Input Code

```
//
// File:   zbt_6111_sample.v
// Date:   26-Nov-05
// Author: I. Chuang
//
// 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  mouse_clock, mouse_data, keyboard_clock, keyboard_data;

input  clock_27mhz, clock1, clock2;

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

```
   output  disp_data_out;

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

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

   inout [43:0] daughtercard;

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

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

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

   // Audio Input and Output
   assign beep= 1'b0;
   assign audio_reset_b = 1'b0;
   assign ac97_synch = 1'b0;
   assign ac97_sdata_out = 1'b0;
/*
*/
   // ac97_sdata_in is an input

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

   // Video Input
   //assign tv_in_i2c_clock = 1'b0;
   assign tv_in_fifo_read = 1'b1;
   assign tv_in_fifo_clock = 1'b0;
   assign tv_in_iso = 1'b1;
   //assign tv_in_reset_b = 1'b0;
   assign tv_in_clock = clock_27mhz;//1'b0;
   //assign tv_in_i2c_data = 1'bZ;
   // tv_in_ycrcb, tv_in_data_valid, tv_in_line_clock1, tv_in_line_clock2,
   // tv_in_aef, tv_in_hff, and tv_in_aff are inputs

   // SRAMs

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

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

/* enable RAM pins */

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

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

   assign ram1_data = 36'hZ;
   assign ram1_address = 19'h0;
   assign ram1_adv_ld = 1'b0;
   assign ram1_clk = 1'b0;
   assign ram1_cen_b = 1'b1;
   assign ram1_ce_b = 1'b1;
   assign ram1_oe_b = 1'b1;
   assign ram1_we_b = 1'b1;
   assign ram1_bwe_b = 4'hF;

   assign clock_feedback_out = 1'b0;
   // clock_feedback_in is an input

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

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

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

   // LED Displays
/*
```

```
   assign disp_blank = 1'b1;
   assign disp_clock = 1'b0;
   assign disp_rs = 1'b0;
   assign disp_ce_b = 1'b1;
   assign disp_reset_b = 1'b0;
   assign disp_data_out = 1'b0;
*/

   // disp_data_in is an input

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

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

   // Daughtercard Connectors
   assign daughtercard = 44'hZ;

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

   // Logic Analyzer

   assign analyzer1_data = 16'h0;
   assign analyzer1_clock = 1'b1;
   assign analyzer2_data = 16'h0;
   assign analyzer2_clock = 1'b1;
   assign analyzer3_data = 16'h0;
   assign analyzer3_clock = 1'b1;
   assign analyzer4_data = 16'h0;
   assign analyzer4_clock = 1'b1;

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


            // Modified to produce 129.6 MHz clock. Divider from 10 to 5
   // use FPGA's digital clock manager to produce a
   // 65MHz clock (actually 64.8MHz)

            wire clock_65mhz, clock_65mhz_unbuf;

   DCM vclk1(.CLKIN(clock_27mhz),.CLKFX(clock_65mhz_unbuf));
   // synthesis attribute CLKFX_DIVIDE of vclk1 is 10
   // synthesis attribute CLKFX_MULTIPLY of vclk1 is 24
// synthesis attribute CLK_FEEDBACK of vclk1 is NONE
// synthesis attribute CLKIN_PERIOD of vclk1 is 37


//BUFG vclk2(.O(clock_65mhz),.I(clock_65mhz_unbuf));
        BUFG vclk3(.O(clock_65mhz),.I(clock_65mhz_unbuf));


        wire fastclk = clock_65mhz;

assign ram0_clk = ~fastclk;



// power-on reset generation
wire power_on_reset;    // remain high for first 16 clocks
SRL16 reset_sr (.D(1'b0), .CLK(fastclk), .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, fastclk, ~button_enter, user_reset);
assign reset = user_reset | power_on_reset;


        // Debouncing and pulse generation

        wire button0_sync, button1_sync, button2_sync, button3_sync;
        wire button_up_sync, button_down_sync;
        wire button_up_pulse, button_down_pulse;

        debounce dbx0 (reset, fastclk, ~button0, button0_sync);
        debounce dbx1 (reset, fastclk, ~button1, button1_sync);
        debounce dbx2 (reset, fastclk, ~button2, button2_sync);
        debounce dbx3 (reset, fastclk, ~button3, button3_sync);
        debounce dbxu (reset, fastclk, ~button_up, button_up_sync);
        debounce dbxd (reset, fastclk, ~button_down, button_down_sync);

        pulser pulse1 (reset, fastclk, button_up_sync, button_up_pulse);
        pulser pulse2 (reset, fastclk, button_down_sync, button_down_pulse);

        wire [7:0] switch_sync;

debounce dbs0 (reset, fastclk, switch[0], switch_sync[0]);
debounce dbs1 (reset, fastclk, switch[1], switch_sync[1]);
debounce dbs2 (reset, fastclk, switch[2], switch_sync[2]);
        debounce dbs3 (reset, fastclk, switch[3], switch_sync[3]);
        debounce dbs4 (reset, fastclk, switch[4], switch_sync[4]);
        debounce dbs5 (reset, fastclk, switch[5], switch_sync[5]);
        debounce dbs6 (reset, fastclk, switch[6], switch_sync[6]);
        debounce dbs7 (reset, fastclk, switch[7], switch_sync[7]);
```

```
// generate basic XVGA video signals
wire [10:0] hcount;
wire [9:0]  vcount;
wire hsync,vsync,blank;
xvgafast xvga1(fastclk, hcount,vcount,hsync,vsync,blank);

// wire up to ZBT ram

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

zbt_6111 zbt1(fastclk, vram_we, vram_addr,
                 vram_write_data, vram_read_data,
                 //ram0_clk,
                       ram0_we_b, ram0_address, ram0_data, ram0_cen_b);

// generate pixel value from reading ZBT memory
wire [23:0]      vr_pixel;
wire [18:0]      vram_addr1;
      wire vr_read;  // Set to high when video display is reading

      wire cutoff = switch_sync[2];
vram_display vd1(reset,fastclk, hcount, vcount,
                             vr_pixel, vram_addr1, vram_read_data, vr_read, cutoff);
      // vr_pixel becomes valid for (hcount,vcount) after a delay of 4.

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

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

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

// code to write NTSC data to video memory

wire [18:0] ntsc_addr;
wire [35:0] ntsc_data;
wire        ntsc_we;
ntsc_to_zbt n2z (fastclk, tv_in_line_clock1, fvh, dv, ycrcb,
                 ntsc_addr, ntsc_data, ntsc_we, 1'b1);


assign  vram_addr = vram_we ? ntsc_addr : vram_addr1;
      assign vram_we = ~switch_sync[1] & ~vr_read;
```

```
   assign  vram_write_data = ntsc_data;



   // select output pixel data

   reg [23:0]          pixel;
   wire    b,hs,vs;



   delayN dn1(fastclk,hsync,hs);
   delayN dn2(fastclk,vsync,vs);
   delayN dn3(fastclk,blank,b);



           wire drive_border, drive_inside;
           wire [10:0] x1, x2;
           wire [9:0] y1, y2;
           assign x1 = 320-8;
           assign y1 = 240-8;

           assign x2 = 320+8;
           assign y2 = 240+8;


   // Drawrectangle and colormatch have 2 clock cycle delay.

   drawrectangle dr1 (reset, fastclk, hcount, vcount, x1, y1, x2, y2, drive_border, drive_inside);


           wire drive_window, drive_window_inside;
           wire [10:0] xx1, xx2;
           wire [9:0] yy1, yy2;
           assign xx1 = 40; assign xx2 = 728;
           assign yy1 = 78; assign yy2 = 500;
           drawrectangle dr2 (reset, fastclk, hcount, vcount, xx1, yy1, xx2, yy2, drive_window,
drive_window_inside);



           wire pmatch, xmatch;


           colormatch cm1 (reset, fastclk, vr_pixel, pmatch, xmatch);






           wire [15:0] xavg1, yavg1;
           wire [15:0] xavg2, yavg2;
```

```
        Locate2D MyCam1 (reset, fastclk, hcount, vcount, pmatch, drive_window_inside,
                                        xavg1, yavg1);

        Locate2D MyCam2 (reset, fastclk, hcount, vcount, xmatch, drive_window_inside,
                                        xavg2, yavg2);


        assign led = 8'hFF;





        wire allmode, crazymode;
        assign allmode = switch_sync[0];
        assign crazymode = switch_sync[2];

        reg centerpoint1, centerpoint;

        always @ (posedge fastclk) begin
                if (reset) begin
                        centerpoint <= 0;
                        centerpoint1 <= 0;
                end else if (hcount > xavg1-4 && hcount < xavg1+4 && vcount > yavg1-4 && vcount <
yavg1+4) begin
                        centerpoint1 <= 1;
                end else if (hcount > xavg2-4 && hcount < xavg2+4 && vcount > yavg2-4 && vcount <
yavg2+4) begin
                        centerpoint1 <= 1;
                end else begin
                        centerpoint1 <= 0;
                end
                centerpoint <= centerpoint1;

        end

        wire [7:0] Rx, Gx, Bx;
        assign Rx = vr_pixel[23:16];
        assign Gx = vr_pixel[15:8];
        assign Bx = vr_pixel[7:0];

        wire [7:0] Rm, Gm, Bm;
        wire [23:0] crazypixel;
        assign Rm = (Rx > 8'd128) ? 8'd255 : 8'd0;
        assign Gm = (Gx > 8'd128) ? 8'd255 : 8'd0;
        assign Bm = (Bx > 8'd128) ? 8'd255 : 8'd0;
        assign crazypixel = {Rm,Gm,Bm};

  always @(posedge fastclk)
   begin
                if (reset) begin
                        pixel <= 24'hFF0000;
                end else begin
                                if (button3_sync)
                                        pixel <= {hcount[8:6],5'b0,hcount[8:6],5'b0,hcount[8:6],5'b0};
                                else if (crazymode) begin
```

```
                                        pixel <= crazypixel;
                                end else if (drive_border)
                                        pixel <= 24'hFFFFFF;
                                else if (centerpoint)
                                        pixel <= 24'hFFFFFF;
                                else if (drive_window)
                                        pixel <= 24'hFFFFFF;
                                else if (allmode)
                                        pixel <= vr_pixel;
                                else if (pmatch || xmatch)
                                        pixel <= vr_pixel;
                                else
                                        pixel <= 24'h000000;

                        end
   end


//assign pixel = 24'hFF0000;


        wire [23:0] avgpixel;
        boxavg bavg (reset, fastclk, hcount, vcount, vsync, vr_pixel, avgpixel);

   // VGA Output.  In order to meet the setup and hold times of the
   // AD7125, we send it ~clock_65mhz.
   assign vga_out_red = pixel[23:16];
   assign vga_out_green = pixel[15:8];
   assign vga_out_blue = pixel[7:0];
   assign vga_out_sync_b = 1'b1;    // not used
        assign vga_out_pixel_clock = ~fastclk;
   assign vga_out_blank_b = ~b;
   assign vga_out_hsync = hs;
   assign vga_out_vsync = vs;

   // debugging


   // display module for debugging

   reg [63:0] dispdata;
   display_16hex hexdisp1(reset, clock_27mhz, dispdata,
                          disp_blank, disp_clock, disp_rs, disp_ce_b,
                          disp_reset_b, disp_data_out);

   always @(posedge clock_27mhz)

         dispdata <= {avgpixel[23:16], 4'b0, avgpixel[15:8], 4'b0, avgpixel[7:0],  32'b0};




endmodule

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

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

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

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

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

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

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

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


module xvgafast(fastclk, hcount,vcount,hsync,vsync,blank);
   input fastclk;
   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 fastclk) begin
           hcount <= hreset ? 0 : hcount + 1;
           hblank <= next_hblank;
           hsync <= hsyncon ? 0 : hsyncoff ? 1 : hsync;  // active low

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

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

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

module vram_display(reset, fastclk, hcount, vcount, vr_pixel,
                    vram_addr, vram_read_data, vr_read, cutoff);

   input reset, fastclk;
   input [10:0] hcount;
   input [9:0]        vcount;
   output [23:0] vr_pixel;
   output [18:0] vram_addr;
   input [35:0]  vram_read_data;
           output vr_read; // Say when we want to read.
```

```
        input cutoff;

        wire [18:0] vram_addr = {1'b0, vcount[8:0], hcount[9:1]};

   reg [35:0]       vr_data_latched;


        reg vr_read;
   always @(posedge fastclk)
              begin
                     if (hcount[0] == 0) begin    // Initiate next read
                            vr_read <= 1;
                     end else if (hcount[0] == 1) begin   // Latch in old data (2 cycles ago)
                            vr_data_latched <= vram_read_data;
                            vr_read <= 0; // Leave memory bus open for NTSC
                     end
              end

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

        reg [23:0] vr_pixel;
        reg [23:0] vr_pixel_latch1;
        reg [23:0] vr_pixel_latch2;

        assign R[7:2] = hcount[0] ? vr_data_latched[35:30] : vr_data_latched[17:12];
        assign G[7:2] = hcount[0] ? vr_data_latched[29:24] : vr_data_latched[11:6];
        assign B[7:2] = hcount[0] ? vr_data_latched[23:18] : vr_data_latched[5:0];

        assign R[1:0] = 2'b0;
        assign G[1:0] = 2'b0;
        assign B[1:0] = 2'b0;


        always @ (posedge fastclk)
              begin
                     if (reset) begin
                            vr_pixel <= 24'b0;
                     end else begin

                            vr_pixel_latch1 <= {R,G,B};
                            vr_pixel_latch2 <= vr_pixel_latch1;
                            vr_pixel <= vr_pixel_latch2;
                     end
              end

endmodule // vram_display

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

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

   parameter NDELAY = 6;
   reg [NDELAY-1:0] shiftreg;
   wire        out = shiftreg[NDELAY-1];

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

endmodule // delayN


module delay1(clk,in,out);
   input clk;
   input in;
   output out;

   parameter NDELAY = 1;

   reg shiftreg;
   wire        out = shiftreg;

   always @(posedge clk)
     shiftreg <= in;

endmodule

module delay2(clk,in,out);
   input clk;
   input in;
   output out;

   parameter NDELAY = 2;

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

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

endmodule

module delay3(clk,in,out);
   input clk;
   input in;
   output out;

   parameter NDELAY = 3;

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

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

endmodule

module delay4(clk,in,out);
```

```
   input clk;
   input in;
   output out;

   parameter NDELAY = 4;

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

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

endmodule




// Produces an average R,G,B value from pixels in the sample box.
// This average is displayed on the LED display for experimentation purposes.
//
// We average over a 16x16 block of pixels, which requires division by 256,
// which can easily be done using a right shift.

module boxavg(reset, fastclk, hcount, vcount, vsync, vr_pixel, avgpixel);
    input reset;
    input fastclk;
    input [10:0] hcount;
    input [9:0] vcount;
               input vsync;
    input [23:0] vr_pixel;
    output [23:0] avgpixel;


               reg [7:0] avgR, avgG, avgB;
               reg [15:0] sumR, sumG, sumB;

               assign avgpixel = {avgR, avgG, avgB};
               reg lastvsync;

               always @ (posedge fastclk) begin
                              lastvsync <= vsync;
                              if (reset) begin
                                             sumR <= 0;
                                             sumG <= 0;
                                             sumB <= 0;
                              end else if (lastvsync == 1 && vsync == 0) begin
                                             avgR <= sumR[15:8];
                                             avgG <= sumG[15:8];
                                             avgB <= sumB[15:8];
                                             sumR <= 0;
                                             sumG <= 0;
                                             sumB <= 0;
                              end else if (hcount >= 320-8 && hcount < 320+8 && vcount >= 240-8 &&
vcount < 240+8) begin
                                             sumR <= sumR + vr_pixel[23:16];
                                             sumG <= sumG + vr_pixel[15:8];
```

```
                                        sumB <= sumB + vr_pixel[7:0];
                                end
        end



endmodule



/*
   colormatch module
                        This module is responsible for isolating the pixels on the screen
                        which most likely correspond to the red and green ankle bands.
                        Each pixel is tested by comparing the ratio of R:G:B values with
                        the reference measured value ranges corresponding to both the red
                        and green ankle bands.

                        To indicate that a pixel matches the red band, "pmatch" is set to high,
                        and to indicate that a pixel matches the green band, "xmatch" is set
                        to high.

                        In addition, this module attempts to improve noise elimination by
                        averaging over twelve pixels with hysteresis. For example, if 6 or more
                        pixels in a block of 12 are red, then the current pixel and later pixels
                        will be flagged as red (pmatch high). This will continue even if the
                        number of pixels which are red drops as low as 2. This is to make it more
                        likely that the solid red band will be picked up strongly, while short
                        noise bursts of red will be ignored.

                        There is a delay between when a sequence of RGB values appear at vr_pixel,
                        and when it         affects the output of pmatch or xmatch. This will shift the image
                        and average measurements over by a few pixels, but the effect is negligible.
 */



module colormatch(reset, fastclk, vr_pixel, pmatch, xmatch);

   input reset;
   input fastclk;
   input [23:0] vr_pixel;
   output pmatch;
         output xmatch;


         wire [7:0] R, G, B;
         assign R = vr_pixel[23:16];
         assign G = vr_pixel[15:8];
         assign B = vr_pixel[7:0];


        reg pmatch;
        reg xmatch;

        reg [11:0] pmatchL, xmatchL;
	reg inred, ingreen;
	wire [4:0] pmatchcount, xmatchcount;


	// Number of Red Pixels in last twelve
	assign pmatchcount = pmatchL[0]  + pmatchL[1] + pmatchL[2] + pmatchL[3] + pmatchL[4] +
								pmatchL[5]  + pmatchL[6] +
pmatchL[7] + pmatchL[8] + pmatchL[9] +
								pmatchL[10] + pmatchL[11];

	// Number of Green Pixels in last twelve
	assign xmatchcount = xmatchL[0]  + xmatchL[1] + xmatchL[2] + xmatchL[3] + xmatchL[4] +
								xmatchL[5]  + xmatchL[6] +
xmatchL[7] + xmatchL[8] + xmatchL[9] +
								xmatchL[10] + xmatchL[11];


	always @ (posedge fastclk)
		if (reset) begin
			pmatchL <= 0;
			pmatch <= 0;
			inred <= 0;
		end else begin

			// Red Ratios: 5B < 10G,  10G < 8B,  10B < 7R,  10G < 6R
			// Red Threshold: R >= 100
			if ((4'd5)*B < (4'd10)*G && (4'd10)*G < (4'd8)*B &&
				(4'd10)*B < (4'd7)*R && (4'd10)*G < (4'd6)*R &&
				R >= 100) begin
					pmatchL <= {pmatchL[10:0], 1'b1};
			end else begin
					pmatchL <= {pmatchL[10:0], 1'b0};
			end

			if (pmatchcount > 5) begin
				inred <= 1;
				pmatch <= 1;
			end else if (inred && pmatchcount >= 2) begin
				inred <= 1;
				pmatch <= 1;
			end else if (inred) begin
				inred <= 0;
				pmatch <= 0;
			end

		end

	always @ (posedge fastclk)
		if (reset) begin
			xmatchL <= 0;
			xmatch <= 0;
			ingreen <= 0;
		end else begin


			// Green Ratios: 10R < 9G, 9B < 10R, 10R < 14B, 13B < 10G
```

```
                    // Threshold for green: G > 71
                    if ((4'd10)*R < (4'd9)*G &&
                            (4'd9)*B < (4'd10)*R &&

                            (4'd10)*R < (4'd14)*B &&
                            (4'd13)*B < (4'd10)*G &&
                             G >= 8'd71) begin
                                xmatchL <= {xmatchL[10:0], 1'b1};  // Shift samples over
                    end else begin
                                xmatchL <= {xmatchL[10:0], 1'b0};
                    end


                    // Greater than 5 is condition for relaxing conditions
                    if (xmatchcount > 5) begin
                            ingreen <= 1;
                            xmatch <= 1;
                    end else if (ingreen && xmatchcount >= 2) begin // Relaxed conditions
                            ingreen <= 1;
                            xmatch <= 1;
                    end else if (ingreen) begin
                            ingreen <= 0;
                            xmatch <= 0;
                    end

            end



endmodule


/*
  Determines when hcount, vcount are inside of a rectangle, or on the border.
  Has 1 clock cycle delay.
 */

module drawrectangle(reset, fastclk, hcount, vcount, x1, y1, x2,
            y2, drive_border, drive_inside);
  input reset;
  input fastclk;
  input [10:0] hcount;
  input [9:0] vcount;
  input [10:0] x1;
  input [9:0] y1;
  input [10:0] x2;
  input [9:0] y2;
  output drive_border, drive_inside;


        reg drive_border, drive_inside;
        reg drive_border1, drive_inside1;

        always @ (posedge fastclk) begin
            if (reset) begin
                    drive_border <= 0;
```

```
                                        drive_inside <= 0;
                                        drive_border1 <= 0;
                                        drive_inside1 <= 0;
                              end else  begin

                                        if ((hcount == x1 || hcount == x2) && vcount >= y1 && vcount <= y2) begin
                                                  drive_border1 <= 1;
                                        end else if ((hcount >= x1 && hcount <= x2) && (vcount == y1 || vcount ==
y2)) begin
                                                  drive_border1 <= 1;
                                        end else begin
                                                  drive_border1 <= 0;
                                        end

                                        if (hcount >= x1 && hcount <= x2 && vcount >= y1 && vcount <= y2) begin
                                                  drive_inside1 <= 1;
                                        end else begin
                                                  drive_inside1 <= 0;
                                        end
                                        drive_border <= drive_border1;
                                        drive_inside <= drive_inside1;

                              end
   end


endmodule

/* Locate2D
   Finds the average position over an entire frame of the pixels
          which satisfy a matching property. (Provided by pmatch).

          The strategy is to sum the X coordinates and Y coordinates of all
          pixels that match, and then divide by the total number we saw.
          This finds the "center of mass" of the distribution.
 */


module Locate2D(reset, fastclk, hcount, vcount, pmatch, drive_window_inside,
                                                  xavg, yavg,
                                                  xsumL, ysumL,
                                                  xcountL, ycountL);
   input reset;
   input fastclk;
   input [10:0] hcount;
  input [9:0] vcount;
   input pmatch;
  input drive_window_inside;
   output [15:0] xavg;
          output [15:0] yavg;

          output [31:0] xsumL, ysumL;
          output [18:0] xcountL, ycountL;


          reg [31:0] xsum, ysum;
          reg [18:0] xcount, ycount;

          reg [31:0] xsumL, ysumL;
          reg [18:0] xcountL, ycountL;



          // Instantiate Divisors to do averaging
          wire aclr1 = 0; wire aclr2 = 0;
          wire sclr1 = 0; wire sclr2 = 0;
          wire ce1 = 1; wire ce2 = 1;

          wire [31:0] dividend1 = xsumL;
          wire [18:0] divisor1 = xcountL;

          wire [31:0] dividend2 = ysumL;
          wire [18:0] divisor2 = ycountL;

          // Outputs
          wire [31:0] quot1, quot2;
          wire [18:0] remd1, remd2;
          wire rfd1, rfd2;


          avgdiv uut1 (dividend1, divisor1, quot1, remd1, fastclk, rfd1, aclr1, sclr1, ce1);
          avgdiv uut2 (dividend2, divisor2, quot2, remd2, fastclk, rfd2, aclr2, sclr2, ce2);

          reg [15:0] xavg, yavg;

          reg doadd, latchsum, latchavg;
          reg [31:0] xadd, yadd;

          always @ (posedge fastclk)
          begin
                  if (reset) begin
                          doadd <= 0;
                          latchsum <= 0;
                          latchavg <= 0;
                          xadd <= 0;
                          yadd <= 0;
                  end else  begin
                          doadd <= 0;
                          latchsum <= 0;
                          latchavg <= 0;

                          if (vcount == 0 && hcount == 1) begin
                                  // Begin division
                                  latchsum <= 1;
                          end else if (vcount == 2 && hcount == 1) begin
                                  // Division must be done by now. Output it.
                                  latchavg <= 1;
                          end else if (drive_window_inside && pmatch) begin
                                  // Trigger addition sequence
                                  doadd <= 1;
                                  xadd <= {21'b0, hcount};
                                  yadd <= {22'b0, vcount};
                          end
                  end
          end

          always @ (posedge fastclk) begin
                  if (reset) begin
                          xsum <= 0;
                          ysum <= 0;
                          xcount <= 0;
                          ycount <= 0;
                          xavg <= 0;
                          yavg <= 0;
                          xsumL <= 0;
                          xcountL <= 0;
                          ysumL <= 0;
                          ycountL <= 0;
                  end else begin
                          if (doadd) begin
                                  xsum <= xsum + xadd;
                                  ysum <= ysum + yadd;
                                  xcount <= xcount + 1;
                                  ycount <= ycount + 1;
                          end else if (latchsum) begin
                                  // Latch and begin division.
                                  xsumL <= xsum;
                                  xcountL <= xcount;
                                  ysumL <= ysum;
                                  ycountL <= ycount;
                                  // Reset for new count.
                                  xsum <= 0;
                                  ysum <= 0;
                                  xcount <= 0;
                                  ycount <= 0;
                          end else if (latchavg) begin
                                  xavg <= quot1[15:0];
                                  yavg <= quot2[15:0];
                          end
                  end
          end

  endmodule

  //
  // File:   ntsc2zbt.v
  // Date:   27-Nov-05
  // Author: I. Chuang
  //
  // 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

/*
   The original source of this code comes from the 6.111 Fall 2005 website.
   Modifications Done For Our Project:
        * YCrCb to RGB conversion added.
             Only 6-bits of resolution are used, for a total of 18-bits per pixel.
        * ZBT Memory writes now are done every two pixels.
        * Both even and odd lines are used.
*/


module ntsc_to_zbt(fastclk, vclk, fvh, dv, ycrcb, ntsc_addr, ntsc_data, ntsc_we, sw);

   input    fastclk;
         input vclk;
   input [2:0]        fvh;
   input    dv;
   input [29:0]       ycrcb;
   output [18:0] ntsc_addr;
   output [35:0] ntsc_data;
   output  ntsc_we;          // write enable for NTSC data
   input    sw;              // switch which determines mode (for debugging)

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



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

   reg [9:0]          col = 0;
   reg [9:0]          row = 0;
   reg [17:0]         vdata = 0;
   reg                vwe;
   reg                old_dv;
   reg                old_frame;        // frames are even / odd interlaced
   reg                even_odd;         // decode interlaced frame to this wire

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

         wire [7:0] R,G,B;
         wire [17:0] din = {R[7:2], G[7:2], B[7:2]};

  slow_YCrCb2RGB torgb (R, G, B, vclk, reset, ycrcb[29:20], ycrcb[19:10], ycrcb[9:0]);
  // For debugging: Comment out above and uncomment below.
  // This enables black and white mode.
  //assign R = ycrcb[29:22];
  //assign G = ycrcb[29:22];
  //assign B = ycrcb[29:22];


  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],y[1:0];
 reg [17:0] data[1:0];
 reg       we[1:0];
 reg           eo[1:0];

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

 // edge detection on write enable signal

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

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



reg [35:0] mydata;
 always @(posedge fastclk)
   if (we_edge)
                                    mydata <= {data[1], mydata[35:18]};


 // compute address to store data in
            wire [18:0] myaddr = {1'b0, y[1][7:0], eo[1], x[1][9:1]};


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

 reg [18:0] ntsc_addr;
 reg [35:0] ntsc_data;
            wire ntsc_we = we_edge & (x[1][0] == 1'b0);
   always @(posedge fastclk)
     if ( ntsc_we )
       begin
                              ntsc_addr <= myaddr;
                              ntsc_data <= mydata;
       end

endmodule // ntsc_to_zbt


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

/* This code was taken from a Xilinx information page */

module slow_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


//
// File:   zbt_6111.v
// Date:   27-Nov-05
// Author: I. Chuang
//
// 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.
///////////////////////////////////////////////////////////////////////////


// - MODIFIED FOR 6.111 FINAL PROJECT -
// Added an extra register for both inputs and outputs.
// Adds an extra two clock cycles to reading, but fixes glitches.


module zbt_6111(clk, weU, addrU, write_dataU, read_dataU,
                ram_we_b, ram_address, ram_data, ram_cen_b);

  input clk;                        // system clock
  input weU;                        // write enable (active HIGH)
  input [18:0] addrU;               // memory address
  input [35:0] write_dataU;         // data to write
  output [35:0] read_dataU;         // data read from memory
  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 = 1'b0;



        // Extra latching steps below.
        reg we;
        reg [18:0] addr;
        reg [35:0] write_data;
        reg [35:0] read_dataU;
        always @ (posedge clk) begin
                we <= weU;
                addr <= addrU;
                write_data <= write_dataU;
                read_dataU <= ram_data;
        end

  reg [1:0]   we_delay;

  always @(posedge clk) begin
         we_delay <= {we_delay[0], we};
        end

  // create two-stage pipeline for write data
```

```
   reg [35:0]  write_data_old1;
   reg [35:0]  write_data_old2;
   always @(posedge clk) begin
      {write_data_old2, write_data_old1} <= {write_data_old1, write_data};
          end
   // wire to ZBT RAM signals

   assign      ram_we_b = ~we;
   assign      ram_address = addr;
   assign      ram_data = we_delay[1] ? write_data_old2 : {36{1'bZ}};


endmodule // zbt_6111


module xvgafast(fastclk, hcount,vcount,hsync,vsync,blank);
   input fastclk;
   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 fastclk) begin
          hcount <= hreset ? 0 : hcount + 1;
          hblank <= next_hblank;
          hsync <= hsyncon ? 0 : hsyncoff ? 1 : hsync;  // active low

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

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



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

module vram_display(reset, fastclk, hcount, vcount, vr_pixel,
                                 vram_addr, vram_read_data, vr_read, cutoff);

   input reset, fastclk;
   input [10:0] hcount;
   input [9:0]        vcount;
   output [23:0] vr_pixel;
   output [18:0] vram_addr;
   input [35:0]  vram_read_data;
		output vr_read; // Say when we want to read.
		input cutoff;

		wire [18:0] vram_addr = {1'b0, vcount[8:0], hcount[9:1]};

   reg [35:0]           vr_data_latched;


		reg vr_read;
   always @(posedge fastclk)
			begin
				if (hcount[0] == 0) begin     // Initiate next read
					vr_read <= 1;
				end else if (hcount[0] == 1) begin    // Latch in old data (2 cycles ago)
					vr_data_latched <= vram_read_data;
					vr_read <= 0; // Leave memory bus open for NTSC
				end
			end

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

		reg [23:0] vr_pixel;
		reg [23:0] vr_pixel_latch1;
		reg [23:0] vr_pixel_latch2;

		assign R[7:2] = hcount[0] ? vr_data_latched[35:30] : vr_data_latched[17:12];
		assign G[7:2] = hcount[0] ? vr_data_latched[29:24] : vr_data_latched[11:6];
		assign B[7:2] = hcount[0] ? vr_data_latched[23:18] : vr_data_latched[5:0];

		assign R[1:0] = 2'b0;
		assign G[1:0] = 2'b0;
```

```
            assign B[1:0] = 2'b0;


            always @ (posedge fastclk)
                        begin
                                    if (reset) begin
                                                vr_pixel <= 24'b0;
                                    end else begin

                                                vr_pixel_latch1 <= {R,G,B};
                                                vr_pixel_latch2 <= vr_pixel_latch1;
                                                vr_pixel <= vr_pixel_latch2;
                                    end
                        end

endmodule // vram_display


// Turns a long signal into a pulse.

module pulser(reset, clock, in, out);
   input reset;
   input clock;
   input in;
   output out;

            reg out;
            reg pulsed;

            always @ (posedge clock)
                        begin
                                    if (reset) begin
                                                out <= 0;
                                                pulsed <= 0;
                                    end else if (pulsed == 0 && in == 1) begin
                                                out <= 1;
                                                pulsed <= 1;
                                    end else if (in == 0) begin
                                                out <= 0;
                                                pulsed <= 0;
                                    end else begin
                                                out <= 0;
                                    end
                        end
endmodule

//
// File:   video_decoder.v
// Date:   31-Oct-05
// Author: J. Castro (MIT 6.111, fall 2005)
//
// This file contains the ntsc_decode and adv7185init modules
//
// These modules are used to grab input NTSC video data from the RCA
// phono jack on the right hand side of the 6.111 labkit (connect
// the camera to the LOWER jack).
```

```
//

///////////////////////////////////////////////////////////////////////////////
//
// NTSC decode - 16-bit CCIR656 decoder
// By Javier Castro
// This module takes a stream of LLC data from the adv7185
// NTSC/PAL video decoder and generates the corresponding pixels,
// that are encoded within the stream, in YCrCb format.

// Make sure that the adv7185 is set to run in 16-bit LLC2 mode.

module ntsc_decode(clk, reset, tv_in_ycrcb, ycrcb, f, v, h, data_valid);

   // clk - line-locked clock (in this case, LLC1 which runs at 27Mhz)
   // reset - system reset
   // tv_in_ycrcb - 10-bit input from chip. should map to pins [19:10]
   // ycrcb - 24 bit luminance and chrominance (8 bits each)
   // f - field: 1 indicates an even field, 0 an odd field
   // v - vertical sync: 1 means vertical sync
   // h - horizontal sync: 1 means horizontal sync

   input clk;
   input reset;
   input [9:0] tv_in_ycrcb; // modified for 10 bit input - should be P[19:10]
   output [29:0] ycrcb;
   output f;
   output v;
   output h;
   output data_valid;
   // output [4:0] state;

   parameter        SYNC_1 = 0;
   parameter        SYNC_2 = 1;
   parameter        SYNC_3 = 2;
   parameter        SAV_f1_cb0 = 3;
   parameter        SAV_f1_y0 = 4;
   parameter        SAV_f1_cr1 = 5;
   parameter        SAV_f1_y1 = 6;
   parameter        EAV_f1 = 7;
   parameter        SAV_VBI_f1 = 8;
   parameter        EAV_VBI_f1 = 9;
   parameter        SAV_f2_cb0 = 10;
   parameter        SAV_f2_y0 = 11;
   parameter        SAV_f2_cr1 = 12;
   parameter        SAV_f2_y1 = 13;
   parameter        EAV_f2 = 14;
   parameter        SAV_VBI_f2 = 15;
   parameter        EAV_VBI_f2 = 16;




   // In the start state, the module doesn't know where
   // in the sequence of pixels, it is looking.
   // Once we determine where to start, the FSM goes through a normal
   // sequence of SAV process_YCrCb EAV... repeat

   // The data stream looks as follows
   // SAV_FF | SAV_00 | SAV_00 | SAV_XY | Cb0 | Y0 | Cr1 | Y1 | Cb2 | Y2 | ... | EAV sequence
   // There are two things we need to do:
   //   1. Find the two SAV blocks (stands for Start Active Video perhaps?)
   //   2. Decode the subsequent data

   reg [4:0]       current_state = 5'h00;
   reg [9:0]       y = 10'h000;  // luminance
   reg [9:0]       cr = 10'h000; // chrominance
   reg [9:0]       cb = 10'h000; // more chrominance

   assign  state = current_state;

   always @ (posedge clk)
     begin
           if (reset)
             begin

             end
           else
             begin
                // these states don't do much except allow us to know where we are in the stream.
                // whenever the synchronization code is seen, go back to the sync_state before
                // transitioning to the new state
                case (current_state)
                  SYNC_1: current_state <= (tv_in_ycrcb == 10'h000) ? SYNC_2 : SYNC_1;
                  SYNC_2: current_state <= (tv_in_ycrcb == 10'h000) ? SYNC_3 : SYNC_1;
                  SYNC_3: current_state <= (tv_in_ycrcb == 10'h200) ? SAV_f1_cb0 :
                                            (tv_in_ycrcb == 10'h274) ? EAV_f1 :
                                            (tv_in_ycrcb == 10'h2ac) ? SAV_VBI_f1 :
                                            (tv_in_ycrcb == 10'h2d8) ? EAV_VBI_f1 :
                                            (tv_in_ycrcb == 10'h31c) ? SAV_f2_cb0 :
                                            (tv_in_ycrcb == 10'h368) ? EAV_f2 :
                                            (tv_in_ycrcb == 10'h3b0) ? SAV_VBI_f2 :
                                            (tv_in_ycrcb == 10'h3c4) ? EAV_VBI_f2 : SYNC_1;

                  SAV_f1_cb0: current_state <= (tv_in_ycrcb == 10'h3ff) ? SYNC_1 : SAV_f1_y0;
                  SAV_f1_y0: current_state <= (tv_in_ycrcb == 10'h3ff) ? SYNC_1 : SAV_f1_cr1;
                  SAV_f1_cr1: current_state <= (tv_in_ycrcb == 10'h3ff) ? SYNC_1 : SAV_f1_y1;
                  SAV_f1_y1: current_state <= (tv_in_ycrcb == 10'h3ff) ? SYNC_1 : SAV_f1_cb0;

                  SAV_f2_cb0: current_state <= (tv_in_ycrcb == 10'h3ff) ? SYNC_1 : SAV_f2_y0;
                  SAV_f2_y0: current_state <= (tv_in_ycrcb == 10'h3ff) ? SYNC_1 : SAV_f2_cr1;
                  SAV_f2_cr1: current_state <= (tv_in_ycrcb == 10'h3ff) ? SYNC_1 : SAV_f2_y1;
                  SAV_f2_y1: current_state <= (tv_in_ycrcb == 10'h3ff) ? SYNC_1 : SAV_f2_cb0;

                  // These states are here in the event that we want to cover these signals
                  // in the future. For now, they just send the state machine back to SYNC_1
                  EAV_f1: current_state <= SYNC_1;
                  SAV_VBI_f1: current_state <= SYNC_1;
                  EAV_VBI_f1: current_state <= SYNC_1;
                  EAV_f2: current_state <= SYNC_1;
                  SAV_VBI_f2: current_state <= SYNC_1;
```

```
                    EAV_VBI_f2: current_state <= SYNC_1;

                  endcase
               end
     end // always @ (posedge clk)

   // implement our decoding mechanism

   wire y_enable;
   wire cr_enable;
   wire cb_enable;

   // if y is coming in, enable the register
   // likewise for cr and cb
   assign y_enable = (current_state == SAV_f1_y0) ||
                         (current_state == SAV_f1_y1) ||
                         (current_state == SAV_f2_y0) ||
                         (current_state == SAV_f2_y1);
   assign cr_enable = (current_state == SAV_f1_cr1) ||
                          (current_state == SAV_f2_cr1);
   assign cb_enable = (current_state == SAV_f1_cb0) ||
                          (current_state == SAV_f2_cb0);

   // f, v, and h only go high when active
   assign {v,h} = (current_state == SYNC_3) ? tv_in_ycrcb[7:6] : 2'b00;

   // data is valid when we have all three values: y, cr, cb
   assign data_valid = y_enable;
   assign ycrcb = {y,cr,cb};

   reg        f = 0;

   always @ (posedge clk)
     begin
          y <= y_enable ? tv_in_ycrcb : y;
          cr <= cr_enable ? tv_in_ycrcb : cr;
          cb <= cb_enable ? tv_in_ycrcb : cb;
          f <= (current_state == SYNC_3) ? tv_in_ycrcb[8] : f;
     end

endmodule



///////////////////////////////////////////////////////////////////////////////
//
// 6.111 FPGA Labkit -- ADV7185 Video Decoder Configuration Init
//
// Created:
// Author: Nathan Ickes
//
///////////////////////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////////////////////
// Register 0
///////////////////////////////////////////////////////////////////////////////
```

```
`define INPUT_SELECT                4'h0
  // 0: CVBS on AIN1 (composite video in)
  // 7: Y on AIN2, C on AIN5 (s-video in)
  // (These are the only configurations supported by the 6.111 labkit hardware)
`define INPUT_MODE                  4'h0
  // 0: Autodetect: NTSC or PAL (BGHID), w/o pedestal
  // 1: Autodetect: NTSC or PAL (BGHID), w/pedestal
  // 2: Autodetect: NTSC or PAL (N), w/o pedestal
  // 3: Autodetect: NTSC or PAL (N), w/pedestal
  // 4: NTSC w/o pedestal
  // 5: NTSC w/pedestal
  // 6: NTSC 4.43 w/o pedestal
  // 7: NTSC 4.43 w/pedestal
  // 8: PAL BGHID w/o pedestal
  // 9: PAL N w/pedestal
  // A: PAL M w/o pedestal
  // B: PAL M w/pedestal
  // C: PAL combination N
  // D: PAL combination N w/pedestal
  // E-F: [Not valid]

`define ADV7185_REGISTER_0 {`INPUT_MODE, `INPUT_SELECT}

///////////////////////////////////////////////////////////////////////////////
// Register 1
///////////////////////////////////////////////////////////////////////////////

`define VIDEO_QUALITY               2'h0
  // 0: Broadcast quality
  // 1: TV quality
  // 2: VCR quality
  // 3: Surveillance quality
`define SQUARE_PIXEL_IN_MODE        1'b0
  // 0: Normal mode
  // 1: Square pixel mode
`define DIFFERENTIAL_INPUT          1'b0
  // 0: Single-ended inputs
  // 1: Differential inputs
`define FOUR_TIMES_SAMPLING         1'b0
  // 0: Standard sampling rate
  // 1: 4x sampling rate (NTSC only)
`define BETACAM                     1'b0
  // 0: Standard video input
  // 1: Betacam video input
`define AUTOMATIC_STARTUP_ENABLE    1'b1
  // 0: Change of input triggers reacquire
  // 1: Change of input does not trigger reacquire

`define ADV7185_REGISTER_1 {`AUTOMATIC_STARTUP_ENABLE, 1'b0, `BETACAM,
`FOUR_TIMES_SAMPLING, `DIFFERENTIAL_INPUT, `SQUARE_PIXEL_IN_MODE,
`VIDEO_QUALITY}

///////////////////////////////////////////////////////////////////////////////
// Register 2
///////////////////////////////////////////////////////////////////////////////
```

```
`define Y_PEAKING_FILTER                    3'h4
  // 0: Composite =  4.5dB,  s-video =  9.25dB
  // 1: Composite =  4.5dB,  s-video =  9.25dB
  // 2: Composite =  4.5dB,  s-video =  5.75dB
  // 3: Composite =  1.25dB, s-video =  3.3dB
  // 4: Composite =  0.0dB,  s-video =  0.0dB
  // 5: Composite = -1.25dB, s-video = -3.0dB
  // 6: Composite = -1.75dB, s-video = -8.0dB
  // 7: Composite = -3.0dB,  s-video = -8.0dB
`define CORING                              2'h0
  // 0: No coring
  // 1: Truncate if Y < black+8
  // 2: Truncate if Y < black+16
  // 3: Truncate if Y < black+32

`define ADV7185_REGISTER_2 {3'b000, `CORING, `Y_PEAKING_FILTER}

///////////////////////////////////////////////////////////////////////////////
// Register 3
///////////////////////////////////////////////////////////////////////////////

`define INTERFACE_SELECT                    2'h0
  // 0: Philips-compatible
  // 1: Broktree API A-compatible
  // 2: Broktree API B-compatible
  // 3: [Not valid]
`define OUTPUT_FORMAT                       4'h0
  // 0: 10-bit @ LLC, 4:2:2 CCIR656
  // 1: 20-bit @ LLC, 4:2:2 CCIR656
  // 2: 16-bit @ LLC, 4:2:2 CCIR656
  // 3: 8-bit @ LLC, 4:2:2 CCIR656
  // 4: 12-bit @ LLC, 4:1:1
  // 5-F: [Not valid]
  // (Note that the 6.111 labkit hardware provides only a 10-bit interface to
  // the ADV7185.)
`define TRISTATE_OUTPUT_DRIVERS             1'b0
  // 0: Drivers tristated when ~OE is high
  // 1: Drivers always tristated
`define VBI_ENABLE                          1'b0
  // 0: Decode lines during vertical blanking interval
  // 1: Decode only active video regions

`define ADV7185_REGISTER_3 {`VBI_ENABLE, `TRISTATE_OUTPUT_DRIVERS,
`OUTPUT_FORMAT, `INTERFACE_SELECT}

///////////////////////////////////////////////////////////////////////////////
// Register 4
///////////////////////////////////////////////////////////////////////////////

`define OUTPUT_DATA_RANGE                   1'b0
  // 0: Output values restricted to CCIR-compliant range
  // 1: Use full output range
`define BT656_TYPE                          1'b0
  // 0: BT656-3-compatible
  // 1: BT656-4-compatible
```

```
`define ADV7185_REGISTER_4 {`BT656_TYPE, 3'b000, 3'b110, `OUTPUT_DATA_RANGE}

///////////////////////////////////////////////////////////////////////////////
// Register 5
///////////////////////////////////////////////////////////////////////////////


`define GENERAL_PURPOSE_OUTPUTS                4'b0000
`define GPO_0_1_ENABLE                         1'b0
  // 0: General purpose outputs 0 and 1 tristated
  // 1: General purpose outputs 0 and 1 enabled
`define GPO_2_3_ENABLE                         1'b0
  // 0: General purpose outputs 2 and 3 tristated
  // 1: General purpose outputs 2 and 3 enabled
`define BLANK_CHROMA_IN_VBI                    1'b1
  // 0: Chroma decoded and output during vertical blanking
  // 1: Chroma blanked during vertical blanking
`define HLOCK_ENABLE                           1'b0
  // 0: GPO 0 is a general purpose output
  // 1: GPO 0 shows HLOCK status

`define ADV7185_REGISTER_5 {`HLOCK_ENABLE, `BLANK_CHROMA_IN_VBI,
`GPO_2_3_ENABLE, `GPO_0_1_ENABLE, `GENERAL_PURPOSE_OUTPUTS}

///////////////////////////////////////////////////////////////////////////////
// Register 7
///////////////////////////////////////////////////////////////////////////////

`define FIFO_FLAG_MARGIN                       5'h10
  // Sets the locations where FIFO almost-full and almost-empty flags are set
`define FIFO_RESET                             1'b0
  // 0: Normal operation
  // 1: Reset FIFO. This bit is automatically cleared
`define AUTOMATIC_FIFO_RESET                   1'b0
  // 0: No automatic reset
  // 1: FIFO is autmatically reset at the end of each video field
`define FIFO_FLAG_SELF_TIME                    1'b1
  // 0: FIFO flags are synchronized to CLKIN
  // 1: FIFO flags are synchronized to internal 27MHz clock

`define ADV7185_REGISTER_7 {`FIFO_FLAG_SELF_TIME, `AUTOMATIC_FIFO_RESET,
`FIFO_RESET, `FIFO_FLAG_MARGIN}

///////////////////////////////////////////////////////////////////////////////
// Register 8
///////////////////////////////////////////////////////////////////////////////

`define INPUT_CONTRAST_ADJUST                  8'h80

`define ADV7185_REGISTER_8 {`INPUT_CONTRAST_ADJUST}

///////////////////////////////////////////////////////////////////////////////
// Register 9
///////////////////////////////////////////////////////////////////////////////
```

```
`define INPUT_SATURATION_ADJUST                    8'h8C

`define ADV7185_REGISTER_9 {`INPUT_SATURATION_ADJUST}

///////////////////////////////////////////////////////////////////////////////
// Register A
///////////////////////////////////////////////////////////////////////////////

`define INPUT_BRIGHTNESS_ADJUST                    8'h00

`define ADV7185_REGISTER_A {`INPUT_BRIGHTNESS_ADJUST}

///////////////////////////////////////////////////////////////////////////////
// Register B
///////////////////////////////////////////////////////////////////////////////

`define INPUT_HUE_ADJUST                           8'h00

`define ADV7185_REGISTER_B {`INPUT_HUE_ADJUST}

///////////////////////////////////////////////////////////////////////////////
// Register C
///////////////////////////////////////////////////////////////////////////////

`define DEFAULT_VALUE_ENABLE                  1'b0
  // 0: Use programmed Y, Cr, and Cb values
  // 1: Use default values
`define DEFAULT_VALUE_AUTOMATIC_ENABLE          1'b0
  // 0: Use programmed Y, Cr, and Cb values
  // 1: Use default values if lock is lost
`define DEFAULT_Y_VALUE                        6'h0C
  // Default Y value

`define ADV7185_REGISTER_C {`DEFAULT_Y_VALUE,
`DEFAULT_VALUE_AUTOMATIC_ENABLE, `DEFAULT_VALUE_ENABLE}

///////////////////////////////////////////////////////////////////////////////
// Register D
///////////////////////////////////////////////////////////////////////////////

`define DEFAULT_CR_VALUE                       4'h8
  // Most-significant four bits of default Cr value
`define DEFAULT_CB_VALUE                       4'h8
  // Most-significant four bits of default Cb value

`define ADV7185_REGISTER_D {`DEFAULT_CB_VALUE, `DEFAULT_CR_VALUE}

///////////////////////////////////////////////////////////////////////////////
// Register E
///////////////////////////////////////////////////////////////////////////////

`define TEMPORAL_DECIMATION_ENABLE             1'b0
  // 0: Disable
  // 1: Enable
`define TEMPORAL_DECIMATION_CONTROL            2'h0
  // 0: Supress frames, start with even field
  // 1: Supress frames, start with odd field
  // 2: Supress even fields only
  // 3: Supress odd fields only
`define TEMPORAL_DECIMATION_RATE              4'h0
  // 0-F: Number of fields/frames to skip

`define ADV7185_REGISTER_E {1'b0, `TEMPORAL_DECIMATION_RATE,
`TEMPORAL_DECIMATION_CONTROL, `TEMPORAL_DECIMATION_ENABLE}

///////////////////////////////////////////////////////////////////////////////
// Register F
///////////////////////////////////////////////////////////////////////////////

`define POWER_SAVE_CONTROL                    2'h0
  // 0: Full operation
  // 1: CVBS only
  // 2: Digital only
  // 3: Power save mode
`define POWER_DOWN_SOURCE_PRIORITY            1'b0
  // 0: Power-down pin has priority
  // 1: Power-down control bit has priority
`define POWER_DOWN_REFERENCE                  1'b0
  // 0: Reference is functional
  // 1: Reference is powered down
`define POWER_DOWN_LLC_GENERATOR              1'b0
  // 0: LLC generator is functional
  // 1: LLC generator is powered down
`define POWER_DOWN_CHIP                       1'b0
  // 0: Chip is functional
  // 1: Input pads disabled and clocks stopped
`define TIMING_REACQUIRE                      1'b0
  // 0: Normal operation
  // 1: Reacquire video signal (bit will automatically reset)
`define RESET_CHIP                            1'b0
  // 0: Normal operation
  // 1: Reset digital core and I2C interface (bit will automatically reset)

`define ADV7185_REGISTER_F {`RESET_CHIP, `TIMING_REACQUIRE, `POWER_DOWN_CHIP,
`POWER_DOWN_LLC_GENERATOR, `POWER_DOWN_REFERENCE,
`POWER_DOWN_SOURCE_PRIORITY, `POWER_SAVE_CONTROL}

///////////////////////////////////////////////////////////////////////////////
// Register 33
///////////////////////////////////////////////////////////////////////////////

`define PEAK_WHITE_UPDATE                     1'b1
  // 0: Update gain once per line
  // 1: Update gain once per field
`define AVERAGE_BIRIGHTNESS_LINES             1'b1
  // 0: Use lines 33 to 310
  // 1: Use lines 33 to 270
`define MAXIMUM_IRE                           3'h0
  // 0: PAL: 133, NTSC: 122
  // 1: PAL: 125, NTSC: 115
  // 2: PAL: 120, NTSC: 110
  // 3: PAL: 115, NTSC: 105
  // 4: PAL: 110, NTSC: 100
  // 5: PAL: 105, NTSC: 100
  // 6-7: PAL: 100, NTSC: 100
`define COLOR_KILL                              1'b1
  // 0: Disable color kill
  // 1: Enable color kill

`define ADV7185_REGISTER_33 {1'b1, `COLOR_KILL, 1'b1, `MAXIMUM_IRE,
`AVERAGE_BIRIGHTNESS_LINES, `PEAK_WHITE_UPDATE}

`define ADV7185_REGISTER_10 8'h00
`define ADV7185_REGISTER_11 8'h00
`define ADV7185_REGISTER_12 8'h00
`define ADV7185_REGISTER_13 8'h45
`define ADV7185_REGISTER_14 8'h18
`define ADV7185_REGISTER_15 8'h60
`define ADV7185_REGISTER_16 8'h00
`define ADV7185_REGISTER_17 8'h01
`define ADV7185_REGISTER_18 8'h00
`define ADV7185_REGISTER_19 8'h10
`define ADV7185_REGISTER_1A 8'h10
`define ADV7185_REGISTER_1B 8'hF0
`define ADV7185_REGISTER_1C 8'h16
`define ADV7185_REGISTER_1D 8'h01
`define ADV7185_REGISTER_1E 8'h00
`define ADV7185_REGISTER_1F 8'h3D
`define ADV7185_REGISTER_20 8'hD0
`define ADV7185_REGISTER_21 8'h09
`define ADV7185_REGISTER_22 8'h8C
`define ADV7185_REGISTER_23 8'hE2
`define ADV7185_REGISTER_24 8'h1F
`define ADV7185_REGISTER_25 8'h07
`define ADV7185_REGISTER_26 8'hC2
`define ADV7185_REGISTER_27 8'h58
`define ADV7185_REGISTER_28 8'h3C
`define ADV7185_REGISTER_29 8'h00
`define ADV7185_REGISTER_2A 8'h00
`define ADV7185_REGISTER_2B 8'hA0
`define ADV7185_REGISTER_2C 8'hCE
`define ADV7185_REGISTER_2D 8'hF0
`define ADV7185_REGISTER_2E 8'h00
`define ADV7185_REGISTER_2F 8'hF0
`define ADV7185_REGISTER_30 8'h00
`define ADV7185_REGISTER_31 8'h70
`define ADV7185_REGISTER_32 8'h00
`define ADV7185_REGISTER_34 8'h0F
`define ADV7185_REGISTER_35 8'h01
`define ADV7185_REGISTER_36 8'h00
`define ADV7185_REGISTER_37 8'h00
`define ADV7185_REGISTER_38 8'h00
`define ADV7185_REGISTER_39 8'h00
`define ADV7185_REGISTER_3A 8'h00
`define ADV7185_REGISTER_3B 8'h00

`define ADV7185_REGISTER_44 8'h41
`define ADV7185_REGISTER_45 8'hBB
```

```
`define ADV7185_REGISTER_F1 8'hEF
`define ADV7185_REGISTER_F2 8'h80


module adv7185init (reset, clock_27mhz, source, tv_in_reset_b,
                    tv_in_i2c_clock, tv_in_i2c_data);

  input reset;
  input clock_27mhz;
  output tv_in_reset_b; // Reset signal to ADV7185
  output tv_in_i2c_clock; // I2C clock output to ADV7185
  output tv_in_i2c_data; // I2C data line to ADV7185
  input source; // 0: composite, 1: s-video

  initial begin
    $display("ADV7185 Initialization values:");
    $display("  Register 0:  0x%X", `ADV7185_REGISTER_0);
    $display("  Register 1:  0x%X", `ADV7185_REGISTER_1);
    $display("  Register 2:  0x%X", `ADV7185_REGISTER_2);
    $display("  Register 3:  0x%X", `ADV7185_REGISTER_3);
    $display("  Register 4:  0x%X", `ADV7185_REGISTER_4);
    $display("  Register 5:  0x%X", `ADV7185_REGISTER_5);
    $display("  Register 7:  0x%X", `ADV7185_REGISTER_7);
    $display("  Register 8:  0x%X", `ADV7185_REGISTER_8);
    $display("  Register 9:  0x%X", `ADV7185_REGISTER_9);
    $display("  Register A:  0x%X", `ADV7185_REGISTER_A);
    $display("  Register B:  0x%X", `ADV7185_REGISTER_B);
    $display("  Register C:  0x%X", `ADV7185_REGISTER_C);
    $display("  Register D:  0x%X", `ADV7185_REGISTER_D);
    $display("  Register E:  0x%X", `ADV7185_REGISTER_E);
    $display("  Register F:  0x%X", `ADV7185_REGISTER_F);
    $display("  Register 33: 0x%X", `ADV7185_REGISTER_33);
  end

  //
  // Generate a 1MHz for the I2C driver (resulting I2C clock rate is 250kHz)
  //

  reg [7:0] clk_div_count, reset_count;
  reg clock_slow;
  wire reset_slow;

  initial
   begin
        clk_div_count <= 8'h00;
        // synthesis attribute init of clk_div_count is "00";
        clock_slow <= 1'b0;
        // synthesis attribute init of clock_slow is "0";
   end

  always @(posedge clock_27mhz)
   if (clk_div_count == 26)
    begin
         clock_slow <= ~clock_slow;
         clk_div_count <= 0;
    end
  else
    clk_div_count <= clk_div_count+1;

always @(posedge clock_27mhz)
  if (reset)
    reset_count <= 100;
  else
    reset_count <= (reset_count==0) ? 0 : reset_count-1;

assign reset_slow = reset_count != 0;

//
// I2C driver
//

reg load;
reg [7:0] data;
wire ack, idle;

i2c i2c(.reset(reset_slow), .clock4x(clock_slow), .data(data), .load(load),
              .ack(ack), .idle(idle), .scl(tv_in_i2c_clock),
              .sda(tv_in_i2c_data));

//
// State machine
//

reg [7:0] state;
reg tv_in_reset_b;
reg old_source;

always @(posedge clock_slow)
   if (reset_slow)
           begin
              state <= 0;
              load <= 0;
              tv_in_reset_b <= 0;
              old_source <= 0;
           end
   else
           case (state)
             8'h00:
               begin
                  // Assert reset
                  load <= 1'b0;
                  tv_in_reset_b <= 1'b0;
                  if (!ack)
                          state <= state+1;
               end
             8'h01:
               state <= state+1;
             8'h02:
               begin
                  // Release reset
                  tv_in_reset_b <= 1'b1;
```

```
     state <= state+1;
                end
8'h03:
  begin
     // Send ADV7185 address
     data <= 8'h8A;
     load <= 1'b1;
     if (ack)
             state <= state+1;
  end
8'h04:
  begin
     // Send subaddress of first register
     data <= 8'h00;
     if (ack)
             state <= state+1;
  end
8'h05:
  begin
     // Write to register 0
     data <= `ADV7185_REGISTER_0 | {5'h00, {3{source}}};
     if (ack)
             state <= state+1;
  end
8'h06:
  begin
     // Write to register 1
     data <= `ADV7185_REGISTER_1;
     if (ack)
             state <= state+1;
  end
8'h07:
  begin
     // Write to register 2
     data <= `ADV7185_REGISTER_2;
     if (ack)
             state <= state+1;
  end
8'h08:
  begin
     // Write to register 3
     data <= `ADV7185_REGISTER_3;
     if (ack)
             state <= state+1;
  end
8'h09:
  begin
     // Write to register 4
     data <= `ADV7185_REGISTER_4;
     if (ack)
             state <= state+1;
  end
8'h0A:
  begin
     // Write to register 5
     data <= `ADV7185_REGISTER_5;
     if (ack)
             state <= state+1;
  end
8'h0B:
  begin
     // Write to register 6
     data <= 8'h00; // Reserved register, write all zeros
     if (ack)
             state <= state+1;
  end
8'h0C:
  begin
     // Write to register 7
     data <= `ADV7185_REGISTER_7;
     if (ack)
             state <= state+1;
  end
8'h0D:
  begin
     // Write to register 8
     data <= `ADV7185_REGISTER_8;
     if (ack)
             state <= state+1;
  end
8'h0E:
  begin
     // Write to register 9
     data <= `ADV7185_REGISTER_9;
     if (ack)
             state <= state+1;
  end
8'h0F: begin
   // Write to register A
   data <= `ADV7185_REGISTER_A;
 if (ack)
   state <= state+1;
end
8'h10:
  begin
     // Write to register B
     data <= `ADV7185_REGISTER_B;
     if (ack)
             state <= state+1;
  end
8'h11:
  begin
     // Write to register C
     data <= `ADV7185_REGISTER_C;
     if (ack)
             state <= state+1;
  end
8'h12:
  begin
     // Write to register D
     data <= `ADV7185_REGISTER_D;
     if (ack)
```

```
              state <= state+1;
   end
8'h13:
   begin
      // Write to register E
      data <= `ADV7185_REGISTER_E;
      if (ack)
              state <= state+1;
   end
8'h14:
   begin
      // Write to register F
      data <= `ADV7185_REGISTER_F;
      if (ack)
              state <= state+1;
   end
8'h15:
   begin
      // Wait for I2C transmitter to finish
      load <= 1'b0;
      if (idle)
              state <= state+1;
   end
8'h16:
   begin
      // Write address
      data <= 8'h8A;
      load <= 1'b1;
      if (ack)
              state <= state+1;
   end
8'h17:
   begin
      data <= 8'h33;
      if (ack)
              state <= state+1;
   end
8'h18:
   begin
      data <= `ADV7185_REGISTER_33;
      if (ack)
              state <= state+1;
   end
8'h19:
   begin
      load <= 1'b0;
      if (idle)
              state <= state+1;
   end

8'h1A: begin
    data <= 8'h8A;
    load <= 1'b1;
    if (ack)
      state <= state+1;
end
```

```
8'h1B:
  begin
     data <= 8'h33;
     if (ack)
             state <= state+1;
  end
8'h1C:
  begin
     load <= 1'b0;
     if (idle)
             state <= state+1;
  end
8'h1D:
  begin
     load <= 1'b1;
     data <= 8'h8B;
     if (ack)
             state <= state+1;
  end
8'h1E:
  begin
     data <= 8'hFF;
     if (ack)
             state <= state+1;
  end
8'h1F:
  begin
     load <= 1'b0;
     if (idle)
             state <= state+1;
  end
8'h20:
  begin
     // Idle
     if (old_source != source) state <= state+1;
     old_source <= source;
  end
8'h21: begin
   // Send ADV7185 address
   data <= 8'h8A;
   load <= 1'b1;
   if (ack) state <= state+1;
end
8'h22: begin
   // Send subaddress of register 0
   data <= 8'h00;
   if (ack) state <= state+1;
end
8'h23: begin
   // Write to register 0
   data <= `ADV7185_REGISTER_0 | {5'h00, {3{source}}};
   if (ack) state <= state+1;
end
8'h24: begin
   // Wait for I2C transmitter to finish
   load <= 1'b0;
```

```
                    if (idle) state <= 8'h20;
                 end
       endcase

endmodule

// i2c module for use with the ADV7185

module i2c (reset, clock4x, data, load, idle, ack, scl, sda);

   input reset;
   input clock4x;
   input [7:0] data;
   input load;
   output ack;
   output idle;
   output scl;
   output sda;

   reg [7:0] ldata;
   reg ack, idle;
   reg scl;
   reg sdai;

   reg [7:0] state;

   assign sda = sdai ? 1'bZ : 1'b0;

   always @(posedge clock4x)
     if (reset)
       begin
                 state <= 0;
                 ack <= 0;
       end
     else
       case (state)
                8'h00: // idle
                  begin
                     scl <= 1'b1;
                     sdai <= 1'b1;
                     ack <= 1'b0;
                     idle <= 1'b1;
                     if (load)
                             begin
                                ldata <= data;
                                ack <= 1'b1;
                                state <= state+1;
                             end
                  end
                8'h01: // Start
                  begin
                     ack <= 1'b0;
                     idle <= 1'b0;
                     sdai <= 1'b0;
                     state <= state+1;
                  end
```

```
8'h02:
  begin
     scl <= 1'b0;
     state <= state+1;
  end
8'h03: // Send bit 7
  begin
     ack <= 1'b0;
     sdai <= ldata[7];
     state <= state+1;
  end
8'h04:
  begin
     scl <= 1'b1;
     state <= state+1;
  end
8'h05:
  begin
     state <= state+1;
  end
8'h06:
  begin
     scl <= 1'b0;
     state <= state+1;
  end
8'h07:
  begin
     sdai <= ldata[6];
     state <= state+1;
  end
8'h08:
  begin
     scl <= 1'b1;
     state <= state+1;
  end
8'h09:
  begin
     state <= state+1;
  end
8'h0A:
  begin
     scl <= 1'b0;
     state <= state+1;
  end
8'h0B:
  begin
     sdai <= ldata[5];
     state <= state+1;
  end
8'h0C:
  begin
     scl <= 1'b1;
     state <= state+1;
  end
8'h0D:
  begin
```

```
     state <= state+1;
  end
8'h0E:
  begin
     scl <= 1'b0;
     state <= state+1;
  end
8'h0F:
  begin
     sdai <= ldata[4];
     state <= state+1;
  end
8'h10:
  begin
     scl <= 1'b1;
     state <= state+1;
  end
8'h11:
  begin
     state <= state+1;
  end
8'h12:
  begin
     scl <= 1'b0;
     state <= state+1;
  end
8'h13:
  begin
     sdai <= ldata[3];
     state <= state+1;
  end
8'h14:
  begin
     scl <= 1'b1;
     state <= state+1;
  end
8'h15:
  begin
     state <= state+1;
  end
8'h16:
  begin
     scl <= 1'b0;
     state <= state+1;
  end
8'h17:
  begin
     sdai <= ldata[2];
     state <= state+1;
  end
8'h18:
  begin
     scl <= 1'b1;
     state <= state+1;
  end
8'h19:
  begin
     state <= state+1;
  end
8'h1A:
  begin
     scl <= 1'b0;
     state <= state+1;
  end
8'h1B:
  begin
     sdai <= ldata[1];
     state <= state+1;
  end
8'h1C:
  begin
     scl <= 1'b1;
     state <= state+1;
  end
8'h1D:
  begin
     state <= state+1;
  end
8'h1E:
  begin
     scl <= 1'b0;
     state <= state+1;
  end
8'h1F:
  begin
     sdai <= ldata[0];
     state <= state+1;
  end
8'h20:
  begin
     scl <= 1'b1;
     state <= state+1;
  end
8'h21:
  begin
     state <= state+1;
  end
8'h22:
  begin
     scl <= 1'b0;
     state <= state+1;
  end
8'h23: // Acknowledge bit
  begin
     state <= state+1;
  end
8'h24:
  begin
     scl <= 1'b1;
     state <= state+1;
  end
8'h25:
```

```
                begin
                   state <= state+1;
                end
              8'h26:
                begin
                   scl <= 1'b0;
                   if (load)
                           begin
                              ldata <= data;
                              ack <= 1'b1;
                              state <= 3;
                           end
                   else
                           state <= state+1;
                end
              8'h27:
                begin
                   sdai <= 1'b0;
                   state <= state+1;
                end
              8'h28:
                begin
                   scl <= 1'b1;
                   state <= state+1;
                end
              8'h29:
                begin
                   sdai <= 1'b1;
                   state <= 0;
                end
        endcase

endmodule
```

Game Mode

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    23:31:10 05/01/06
// Design Name:
// Module Name:    draw_game_background
// Project Name:
// Target Device:
// Tool versions:
// Description:
//
// Dependencies:
//
```

```
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
module draw_game_background(reset, pixel_clock, pixel_count, line_count,
                                                                        RGB_in, keyhit_now,
keyhit_next,score,
                                                                        vga_out_red,
vga_out_green, vga_out_blue);

input reset, pixel_clock;

input [10:0] pixel_count;
input [9:0] line_count;

input [23:0] RGB_in;

input [35:0] keyhit_now, keyhit_next;

input [15:0] score;   //the current score of user

output [7:0] vga_out_red, vga_out_green, vga_out_blue;
reg [7:0] vga_out_red, vga_out_green, vga_out_blue;

wire [20:0] white_border1, white_inside1;
wire [14:0] black_border1, black_inside1;
wire [20:0] white_border2, white_inside2;
wire [14:0] black_border2, black_inside2;
wire char_on_title, score_border_on, in_score_board, score_on;
//wire [7:0] red1, green1, blue1;
//wire [7:0] red2, green2, blue2;
//wire [7:0] red3, green3, blue3;
wire pixel;

// displaying the piano board on the left, which shows the current key that is
// being stepped on
defparam row10.start_pixel = 80;
defparam row11.start_pixel = 80;
defparam row12.start_pixel = 80;

defparam row10.start_line = 406;
defparam row11.start_line = 406;
defparam row12.start_line = 406;

draw_game_row row10 (reset, pixel_clock, 0, pixel_count, line_count,
                                                white_border1[6:0], white_inside1[6:0],
                                                black_border1[4:0], black_inside1[4:0]);
draw_game_row row11 (reset, pixel_clock, 1, pixel_count, line_count,
                                                white_border1[13:7], white_inside1[13:7],
                                                black_border1[9:5], black_inside1[9:5]);
draw_game_row row12 (reset, pixel_clock, 2, pixel_count, line_count,
                                                white_border1[20:14], white_inside1[20:14],
                                                black_border1[14:10], black_inside1[14:10]);

// displaying the piano board on the right, which shows the next keys that need
// to be stepped on
defparam row20.start_pixel = 600;
defparam row21.start_pixel = 600;
defparam row22.start_pixel = 600;

defparam row20.start_line = 406;
defparam row21.start_line = 406;
defparam row22.start_line = 406;

draw_game_row row20 (reset, pixel_clock, 0, pixel_count, line_count,
                                        white_border2[6:0], white_inside2[6:0],
                                        black_border2[4:0], black_inside2[4:0]);
draw_game_row row21 (reset, pixel_clock, 1, pixel_count, line_count,
                                        white_border2[13:7], white_inside2[13:7],
                                        black_border2[9:5], black_inside2[9:5]);
draw_game_row row22 (reset, pixel_clock, 2, pixel_count, line_count,
                                        white_border2[20:14], white_inside2[20:14],
                                        black_border2[14:10], black_inside2[14:10]);

// display the title
write_title title (reset, pixel_clock, pixel_count, line_count,
                                       "PIANO DANCE REVOLUTION", char_on_title);

//rectangle score_border (reset, pixel_clock, pixel_count, line_count,
//                                                  11'd600, 10'd150, 11'd800,
10'd300,
//                                                  score_border_on, in_score_board);

// display score
draw_score_board score_board (reset, pixel_clock, pixel_count, line_count,
                                                                       "15", score_on);


// display the Charlie's Angels background
vga_romdisp disp (pixel_clock, pixel_count, line_count, pixel_clock, pixel);

// figuring out the colors needs to be displayed for a given
// pixel for the background
// purple
parameter [7:0] red1 = 8'b01111111;
parameter [7:0] blue1  = 8'b11111111;
parameter [7:0] green1 = 8'd0;

// pink
parameter [7:0] red2 = 8'b11111111;
parameter [7:0] blue2  = 8'b11111111;
parameter [7:0] green2 = 8'd0;

// MIT gray
parameter [7:0] red3 = 8'b01001111;
parameter [7:0] blue3  = 8'b00111111;
parameter [7:0] green3 = 8'b01001111;

always @ (posedge pixel_clock)     // assign color values based on
begin                                                                      // pixels
      if (reset)
```

```
                        begin
                                vga_out_red <= 8'd0;
                                vga_out_green <= 8'd0;
                                vga_out_blue <= 8'd0;
                        end
                else
                        begin
                                if (char_on_title)
                                        begin
                                                vga_out_red <= 8'd0;
                                                vga_out_green <= 8'd0;
                                                vga_out_blue <= 8'd0;
                                        end
                                else if (/*score_border_on ||*/ score_on)
                                        begin
                                                vga_out_red <= 8'b11111111;
                                                vga_out_green <= 8'b11111111;
                                                vga_out_blue <= 8'b11111111;
                                        end
                                else
                                        begin
                                                if ((keyhit_now[35:21] & black_inside1[14:0] |
                                                         keyhit_next[35:21] & black_inside2[14:0]) != 0)
                                                        begin
                                                                vga_out_red <= RGB_in[7:0];
                                                                vga_out_green <= RGB_in[15:8];
                                                                vga_out_blue <= RGB_in[23:16];
                                                        end
                                                else if ((black_border1 != 15'd0) || (black_inside1 != 15'd0) ||
                                                                        (black_border2 != 15'd0) ||
        (black_inside2 != 15'd0))
                                                        begin
                                                                vga_out_red <= 8'd0;
                                                                vga_out_green <= 8'd0;
                                                                vga_out_blue <= 8'd0;
                                                        end
                                                else if ((keyhit_now[20:0] & white_inside1[20:0] |
                                                                        keyhit_next[20:0] &
        white_inside2[20:0]) != 0)
                                                        begin
                                                                vga_out_red <= RGB_in[7:0];
                                                                vga_out_green <= RGB_in[15:8];
                                                                vga_out_blue <= RGB_in[23:16];
                                                        end
                                                else if (white_border1 != 21'd0 || white_border2 != 21'd0)
                                                        begin
                                                                vga_out_red <= 8'd0;
                                                                vga_out_green <= 8'd0;
                                                                vga_out_blue <= 8'd0;
                                                        end
                                                else if (white_inside1 != 21'd0 || white_inside2 != 21'd0)
                                                        begin
                                                                vga_out_red <= 8'b11111111;
                                                                vga_out_green <= 8'b11111111;
                                                                vga_out_blue <= 8'b11111111;
                                                        end
								else if (pixel)
									begin
										if (keyhit_now != 36'd0)
											begin
												vga_out_red <= red2;
												vga_out_green <= green2;
												vga_out_blue <= blue2;
											end
										else
											begin
												vga_out_red <= red1;
												vga_out_green <= green1;
												vga_out_blue <= blue1;
											end
									end
								else
									begin
										if (keyhit_next != 36'd0)
											begin
												vga_out_red <= red3;
												vga_out_green <= green3;
												vga_out_blue <= blue3;
											end
										else
											begin
												vga_out_red <= 0;
												vga_out_green <= 0;
												vga_out_blue <= 0;
											end
									end
							end
					end
end

endmodule




`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    23:41:40 05/01/06
// Design Name:
// Module Name:    draw_game_row
// Project Name:
// Target Device:
// Tool versions:
// Description:
//
// Dependencies:   similar to the piano keyboard, except with different starting
//                                                  value (set by the parameters)
```

```
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
module draw_game_row(reset, pixel_clock, keyrow, pixel_count, line_count,
                                                        wborder, winside, bborder, binside);

parameter [10:0] start_pixel = 80;
parameter [9:0] start_line = 406;

parameter [10:0] black_start_pixel = start_pixel + 35;
parameter [10:0] white_start_pixel = start_pixel;

defparam wk0.start_pixel = white_start_pixel;
defparam wk1.start_pixel = white_start_pixel;
defparam wk2.start_pixel = white_start_pixel;
defparam wk3.start_pixel = white_start_pixel;
defparam wk4.start_pixel = white_start_pixel;
defparam wk5.start_pixel = white_start_pixel;
defparam wk6.start_pixel = white_start_pixel;

defparam wk0.start_line = start_line;
defparam wk1.start_line = start_line;
defparam wk2.start_line = start_line;
defparam wk3.start_line = start_line;
defparam wk4.start_line = start_line;
defparam wk5.start_line = start_line;
defparam wk6.start_line = start_line;

defparam bk0.start_pixel = black_start_pixel;
defparam bk1.start_pixel = black_start_pixel;
defparam bk2.start_pixel = black_start_pixel;
defparam bk3.start_pixel = black_start_pixel;
defparam bk4.start_pixel = black_start_pixel;

defparam bk0.start_line = start_line;
defparam bk1.start_line = start_line;
defparam bk2.start_line = start_line;
defparam bk3.start_line = start_line;
defparam bk4.start_line = start_line;

input reset, pixel_clock;
input [1:0] keyrow;
input [10:0] pixel_count;
input [9:0] line_count;

output [6:0] wborder, winside;
output [4:0] bborder, binside;

game_white_key wk0 (reset, pixel_clock, keyrow, 0, pixel_count, line_count,
                                        wborder[0], winside[0]);
game_white_key wk1 (reset, pixel_clock, keyrow, 1, pixel_count, line_count,
                                        wborder[1], winside[1]);
game_white_key wk2 (reset, pixel_clock, keyrow, 2, pixel_count, line_count,
                                                        wborder[2], winside[2]);
game_white_key wk3 (reset, pixel_clock, keyrow, 3, pixel_count, line_count,
                                                        wborder[3], winside[3]);
game_white_key wk4 (reset, pixel_clock, keyrow, 4, pixel_count, line_count,
                                                 wborder[4], winside[4]);
game_white_key wk5 (reset, pixel_clock, keyrow, 5, pixel_count, line_count,
                                                        wborder[5], winside[5]);
game_white_key wk6 (reset, pixel_clock, keyrow, 6, pixel_count, line_count,
                                                        wborder[6], winside[6]);
game_black_key bk0 (reset, pixel_clock, keyrow, 0, pixel_count, line_count,
                                                        bborder[0], binside[0]);
game_black_key bk1 (reset, pixel_clock, keyrow, 1, pixel_count, line_count,
                                                        bborder[1], binside[1]);
game_black_key bk2 (reset, pixel_clock, keyrow, 2, pixel_count, line_count,
                                                        bborder[2], binside[2]);
game_black_key bk3 (reset, pixel_clock, keyrow, 3, pixel_count, line_count,
                                                        bborder[3], binside[3]);
game_black_key bk4 (reset, pixel_clock, keyrow, 4, pixel_count, line_count,
                                                        bborder[4], binside[4]);

endmodule






`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    23:21:08 05/02/06
// Design Name:
// Module Name:    draw_score_board
// Project Name:
// Target Device:
// Tool versions:
// Description:
//
// Dependencies:   display the screen onto the background
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
module draw_score_board(reset, pixel_clock, pixel_count, line_count, score, score_on);

parameter NumChar = 3;    // number of characters in title
parameter NumCharBit = 2;              // numbers of bits in NumChar

input reset, pixel_clock;
input [10:0] pixel_count;
input [9:0] line_count;
input [NumChar*8-1:0] score;
```

```
output score_on;
reg score_on;

defparam show_score_title.NCHAR = 5;
defparam show_score_title.NCHAR_BITS = 3;
defparam show_score.NCHAR = NumChar;
defparam show_score.NCHAR_BITS = NumCharBit;

wire score_on_1, score_on_2;

char_string_display show_score_title (pixel_clock, pixel_count, line_count, score_on_1,
                                                                                    "SCORE",
11'd725, 10'd200);

char_string_display show_score (pixel_clock, pixel_count, line_count, score_on_2,
                                                                                    score, 11'd750,
10'd250);

always @ (posedge pixel_clock)
begin
        if (reset)
                score_on <= 0;
        else
                score_on <= score_on_1 | score_on_2;
end
endmodule





`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    14:30:31 05/02/06
// Design Name:
// Module Name:    game_black_key
// Project Name:
// Target Device:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////

module game_black_key(reset, pixel_clock, keyrow, keycol, pixel_count, line_count,
                                                        border, inside);
```

```
parameter [10:0] start_pixel = 95;
parameter [9:0] start_line = 406;

input reset, pixel_clock;
input [1:0] keyrow;
input [2:0] keycol;
input [10:0] pixel_count;
input [9:0] line_count;

output border, inside;

reg [10:0] top_corner_pixel, bottom_corner_pixel;
reg [9:0] top_corner_line, bottom_corner_line;

rectangle k0 (reset, pixel_clock, pixel_count, line_count,
                        top_corner_pixel, top_corner_line,
                        bottom_corner_pixel, bottom_corner_line,
                        border, inside);

always @ (posedge pixel_clock)
begin
        if (reset)
                begin
                        top_corner_pixel <= 0;
                        top_corner_line <= 0;
                        bottom_corner_pixel <= 0;
                        bottom_corner_line <= 0;
                end
        else
                begin
                        if (keycol <= 1)
                                begin
                                        top_corner_pixel <= 50*keycol + start_pixel;
                                        bottom_corner_pixel <= 50*keycol + start_pixel + 30;
                                end
                        else
                                begin
                                        top_corner_pixel <= 50*(keycol + 1) + start_pixel;
                                        bottom_corner_pixel <= 50*(keycol + 1) + start_pixel + 30;
                                end
                        top_corner_line <= 100*keyrow + start_line;
                        bottom_corner_line <= 100*keyrow + start_line + 60;
                end
end

endmodule




`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company:
```

```
// Engineer:
//
// Create Date:    14:27:54 05/02/06
// Design Name:
// Module Name:    game_white_key
// Project Name:
// Target Device:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
module game_white_key(reset, pixel_clock, keyrow, keycol, pixel_count, line_count,
                                                                    border, inside);

parameter [10:0] start_pixel = 80;
parameter [9:0] start_line = 406;

input reset, pixel_clock;
input [1:0] keyrow;
input [2:0] keycol;
input [10:0] pixel_count;
input [9:0] line_count;

output border, inside;

reg [10:0] top_corner_pixel, bottom_corner_pixel;
reg [9:0] top_corner_line, bottom_corner_line;

rectangle k0 (reset, pixel_clock, pixel_count, line_count,
                                        top_corner_pixel, top_corner_line,
                                        bottom_corner_pixel, bottom_corner_line,
                                        border, inside);

always @ (posedge pixel_clock)
begin
        if (reset)
                begin
                        top_corner_pixel <= 0;
                        top_corner_line <= 0;
                        bottom_corner_pixel <= 0;
                        bottom_corner_line <= 0;
                end
        else
                begin
                        top_corner_pixel <= 50*keycol       + start_pixel;
                        bottom_corner_pixel <= 50*(keycol + 1) + start_pixel;
                        top_corner_line <= 100*keyrow + start_line;
                        bottom_corner_line <= 100*(keyrow + 1) - 1 + start_line;
                end
end
```

```
endmodule





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

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

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

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

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

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

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

	       clock_feedback_out, clock_feedback_in,

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

	       rs232_txd, rs232_rxd, rs232_rts, rs232_cts,

	       mouse_clock, mouse_data, keyboard_clock, keyboard_data,

	       clock_27mhz, clock1, clock2,

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

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

                   switch,

                   led,

                   user1, user2, user3, user4,

                   daughtercard,

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

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

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

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

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

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

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

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

input  clock_feedback_in;
output clock_feedback_out;

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

output rs232_txd, rs232_rts;
```

```
input  rs232_rxd, rs232_cts;

input  mouse_clock, mouse_data, keyboard_clock, keyboard_data;

input  clock_27mhz, clock1, clock2;

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

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

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

inout [43:0] daughtercard;

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

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

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

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

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

// Video Input
assign tv_in_i2c_clock = 1'b0;
assign tv_in_fifo_read = 1'b0;
assign tv_in_fifo_clock = 1'b0;
assign tv_in_iso = 1'b0;
assign tv_in_reset_b = 1'b0;
assign tv_in_clock = 1'b0;
assign tv_in_i2c_data = 1'bZ;

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

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

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

// LED Displays
assign disp_blank = 1'b1;
assign disp_clock = 1'b0;
assign disp_rs = 1'b0;
assign disp_ce_b = 1'b1;
assign disp_reset_b = 1'b0;
assign disp_data_out = 1'b0;

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

// User I/Os
assign user1 = 32'hZ;
assign user2 = 32'hZ;
assign user3 = 32'hZ;
//assign user4[31:8] = 24'hZ;
        assign user4 = 32'hZ;
```

```
   // Daughtercard Connectors
   assign daughtercard = 44'hZ;

   // SystemACE Microprocessor Port
   assign systemace_data = 16'hZ;
   assign systemace_address = 7'h0;
   assign systemace_ce_b = 1'b1;
   assign systemace_we_b = 1'b1;
   assign systemace_oe_b = 1'b1;

   // Logic Analyzer
   assign analyzer1_data = 16'h0;
   assign analyzer1_clock = 1'b1;
   assign analyzer2_data = 16'h0;
   assign analyzer2_clock = 1'b1;
   assign analyzer3_data = 16'h0;
   assign analyzer3_clock = 1'b1;
   assign analyzer4_data = 16'h0;
   assign analyzer4_clock = 1'b1;

   ////////////////////////////////////////////////////////////////////////////
   //
   // Lab 4 Components
   //
   ////////////////////////////////////////////////////////////////////////////

   //
   // Generate a 64.8MHz pixel clock from clock_27mhz
   //

   wire pclk, pixel_clock;
   DCM pixel_clock_dcm (.CLKIN(clock_27mhz), .CLKFX(pclk));
   // synthesis attribute CLKFX_DIVIDE of pixel_clock_dcm is 10
   // synthesis attribute CLKFX_MULTIPLY of pixel_clock_dcm is 24
   // synthesis attribute CLK_FEEDBACK of pixel_clock_dcm is NONE
   BUFG pixel_clock_buf (.I(pclk), .O(pixel_clock));

wire hblank, vblank, hsync, vsync;
wire [10:0] pixel_count;
wire [9:0] line_count;
reg vga_out_hsync, vga_out_vsync;
reg [1:0] counter;
wire reset_sync;
wire up_sync, down_sync;

assign vga_out_pixel_clock = ~pixel_clock;

assign vga_out_blank_b = (hblank && vblank);

assign vga_out_sync_b = 1'b1;

debounce reset (1'b0, pixel_clock, ~button0, reset_sync);

sync_signal_generator gen (reset_sync, pixel_clock, hblank, vblank, hsync, vsync,
                                                                    pixel_count, line_count);
```

```
//checker_board work (reset_sync, pixel_clock, pixel_count, line_count, vga_out_red, vga_out_green,
vga_out_blue);

wire [35:0] keyhit_now, keyhit_next;
assign keyhit_now[0] = switch[0];
assign keyhit_now[1] = switch[1];
assign keyhit_now[2] = switch[2];
assign keyhit_next[3] = switch[3];
assign keyhit_next[4] = switch[4];
assign keyhit_now[21] = switch[5];
assign keyhit_now[22] = switch[6];
assign keyhit_next[23] = switch[7];
assign keyhit_next[24] = ~button1;
assign keyhit_next[25] = ~button2;
assign keyhit_now[20:3] = 0;
assign keyhit_now[35:23]= 0;
assign keyhit_next[2:0] = 0;
assign keyhit_next[22:5] = 0;
assign keyhit_next[35:26] = 0;

//
//
//draw_background_backward draw (reset_sync, pixel_clock, pixel_count, line_count,
//
24'b111111110000000100000000,
//                                                                              keyhit,
//                                                                              vga_out_red,
vga_out_green, vga_out_blue);

draw_game_background draw (reset_sync, pixel_clock, pixel_count, line_count,

        24'b111111110000000100000000,
                                                                       keyhit_now, keyhit_next,
                                                                       vga_out_red,
vga_out_green, vga_out_blue);

/*reg start;
wire out, enable;
wire done;
reg start_count;

always @ (posedge clock_27mhz)
begin
        if (reset_sync)
                begin
                        start_count <= 0;
                        start <= 0;
                end
        else (enable && !start_count)
                begin
                        start <= 1;
                        start_count <= 1;
                end
        else
                start<= 0;
```

```
end

divider divide (clock_27mhz, reset_sync, enable);



send_to test (reset_sync, clock_27mhz, start, 31'b1000100010001000100010001000100010001000, out, done);

assign led[0] = ~start_count;
assign led[1] = ~out;*/

// testing purposes:
/*
assign user4[0] = pixel_clock;
assign user4[1] = hblank;
assign user4[2] = vblank;
assign user4[3] = hsync;
assign user4[4] = vsync;
assign user4[5] = vga_out_hsync;
assign user4[6] = vga_out_vsync;
assign user4[7] = clock_27mhz;
*/

// delaying the sync signals;
reg temp1h, temp1v, temp2h, temp2v;

always @ (posedge pixel_clock)
begin
        temp1h <= hsync;
        temp1v <= vsync;

        temp2h <= temp1h;
        temp2v <= temp1v;

        vga_out_hsync <= temp2h;
        vga_out_vsync <= temp2v;
end



endmodule






`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    12:59:47 03/21/06
// Design Name:
// Module Name:    sync_signal_generator
// Project Name:
```

```
// Target Device:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
module sync_signal_generator (reset, clk, hblank, vblank, hsync, vsync, pixel_count, line_count);

input reset, clk;
output hblank, vblank, hsync, vsync;
output [9:0] line_count;
output [10:0] pixel_count;
reg hblank, vblank, hsync, vsync;
reg [10:0] pixel_count;
reg [9:0] line_count;


/*parameter hblank_high = 640;
parameter h_front_porch = 16;
parameter hsync_low = 96;
parameter h_back_porch = 48;
parameter h_period = 800;
parameter vblank_high = 480;
parameter v_front_porch = 11;
parameter vsync_low = 2;
parameter v_back_porch = 32;
parameter v_period = 525;*/


// testbench purposes only
/*parameter hblank_high = 6;
parameter h_front_porch = 1;
parameter hsync_low = 3;
parameter h_back_porch = 2;
parameter h_period = 12;
parameter vblank_high = 5;
parameter v_front_porch = 1;
parameter vsync_low = 2;
parameter v_back_porch = 3;
parameter v_period = 11;
*/

parameter hblank_high = 1024;
parameter h_front_porch = 24;
parameter hsync_low = 136;
parameter h_back_porch = 160;
parameter h_period = 1344;
parameter vblank_high = 768;
parameter v_front_porch = 3;
parameter vsync_low = 6;
parameter v_back_porch = 29;
parameter v_period = 806;

always @ (posedge clk)
begin
        if (reset)
        begin
                hsync <= 1;
                vsync <= 1;
                hblank <= 1;
                vblank <= 1;
                pixel_count <= 0;
                line_count <= 0;
        end
        else if (pixel_count < (hblank_high - 1))
//      else if (pixel_count < 639)  // switch to this upon implementation
        begin                                                   // should save space
                hblank <= 1;
                hsync <= 1;
                pixel_count <= pixel_count + 1;
        end
        else if (pixel_count < (hblank_high + h_front_porch - 1))
//      else if (pixel_count < 655)
        begin
                hblank <= 0;
                hsync <= 1;
                pixel_count <= pixel_count + 1;
        end
        else if (pixel_count < (hblank_high + h_front_porch + hsync_low - 1))
//      else if (pixel_count < 751)
        begin
                hblank <= 0;
                hsync <= 0;
                pixel_count <= pixel_count + 1;
        end
        else if (pixel_count < (h_period - 1))
//      else if (pixel_count        < 199)
        begin
                hblank <= 0;
                hsync <= 1;
                pixel_count <= pixel_count + 1;
        end
        else
        begin
                pixel_count <= 0;
                hblank <= 1;
                hsync <= 1;
//              if (line_count < 479)
                if (line_count < (vblank_high - 1))
                begin
                        vsync <= 1;
                        vblank <= 1;
                        line_count <= line_count + 1;
                end
                else if (line_count < (vblank_high + v_front_porch - 1))
//              else if (line_count < 490)
                begin
```

```
                        vsync <= 1;
                        vblank <= 0;
                        line_count <= line_count + 1;

                end
                else if (line_count < (vblank_high + v_front_porch + vsync_low - 1))
//              else if (line_count < 492)
                begin
                        vsync <= 0;
                        vblank <= 0;
                        line_count <= line_count + 1;
                end
                else if (line_count < (v_period - 1))
//              else if (line_count < 524)
                begin
                        vsync <= 1;
                        vblank <= 0;
                        line_count <= line_count + 1;
                end
                else
                begin
                        line_count <= 0;
                        vsync <= 1;
                        vblank <= 1;
                end
        end
end

endmodule





`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    18:46:25 05/02/06
// Design Name:
// Module Name:    write_title
// Project Name:
// Target Device:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
```

```
module write_title(reset, pixel_clock, pixel_count, line_count, title, char_on);

parameter NumChar = 22;  // number of characters in title
parameter NumCharBit = 5;               // numbers of bits in NumChar

input reset, pixel_clock;
input [10:0] pixel_count;
input [9:0] line_count;
input [NumChar*8-1:0] title;

output char_on;

defparam show_title.NCHAR = NumChar;
defparam show_title.NCHAR_BITS = NumCharBit;

char_string_display show_title (pixel_clock, pixel_count, line_count, char_on,
                                                                              title, 11'd400,
10'd15);

endmodule




`timescale 1ns / 1ps

// Takes output of Locate2D and determines which


module Locate3D(reset, clock, xcoord1, ycoord1, zcoord1, zcoord2, xcoord, ycoord, zcoord);

input reset, clock;
input [31:0] xcoord1, ycoord1, zcoord1, zcoord2;

output [31:0] zcoord, xcoord, ycoord;

reg [31:0] zcoord;

wire [31:0] xcoord, ycoord, xcoord1, ycoord1;

reg[31:0] zcoordsum;

always @ (zcoord1 or zcoord2)                   // Average the two z coordinates to find THE z coordinate
begin
zcoordsum = zcoord1 + zcoord2;
zcoord = {0, zcoordsum[31:1]};
end

assign xcoord = xcoord1;
assign ycoord = ycoord1;


endmodule
```