Appendix

Top Level Labkit Code
(developed by Nathan Ickes and modified for project)

```
//////////////////////////////////////////////////////////////////////////////////
//
// 6.111 FPGA Labkit -- Audio/Video Test
//
// For Labkit Revision 004
//
// Created: November 3, 2004
// 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_i2c_data, tv_in_fifo_read, tv_in_fifo_clock,
        tv_in_iso, tv_in_reset_b, tv_in_clock;
inout  [35:0] ram0_data;
output [20:0] 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 [20:0] 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 [24:0] 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;
output  disp_data_out;
input  disp_data_in;
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;

//////////////////////////////////////////////////////////////////////////
//
// 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 reset;
SRL16 reset_sr (.D(1'b0), .CLK(clock_27mhz), .Q(reset),
                    .A0(1'b1), .A1(1'b1), .A2(1'b1), .A3(1'b1));
defparam reset_sr.INIT = 16'hFFFF;

//////////////////////////////////////////////////////////////////////////
//
// Audio Input and Output
//
//////////////////////////////////////////////////////////////////////////

wire [3:0] vol;
wire [39:0] vdisp;
wire volume_up, volume_down;

//////////******debugging wires
wire [19:0] left_in_data;

wire fftdone, rdy, busy, ovflo;
wire [9:0] xn_index, xk_index;
wire [15:0] xk_re, xk_im, xn_re, xn_im;
wire start;
wire [10:0] waddress;
wire fwd_inv;
wire resetout, n_fft_we, fwd_inv_we;
wire [4:0] value;
```

```
wire [9:0] scale_sch, max_index;
wire [4:0] n_fft;
wire rom_enable;

audio audio1 (reset, clock_27mhz, audio_reset_b, ac97_sdata_out,
              ac97_sdata_in, ac97_synch, ac97_bit_clock, switch[1:0],
              {vol, 1'b0}, switch[2], left_in_data,
              fftdone, rdy, busy, ovflo, xn_index, xk_index, xk_re, xk_im,
              xn_re, xn_im,
              n_fft, n_fft_we, fwd_inv, fwd_inv_we, scale_sch, scale_sch_we, start, ce,
              waddress, resetout, value, max_index, rom_enable);

beeper beep1 (reset, clock_27mhz, beep, ~button_enter);
debounce vol_up (reset, clock_27mhz, button_up, volume_up);
debounce vol_down (reset, clock_27mhz, button_down, volume_down);
volume vol1 (reset, clock_27mhz, volume_up, volume_down, vol, vdisp);

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

// SRAMs
assign ram0_data = 36'hZ;
assign ram0_address = 21'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 = 21'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 = 15'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;

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

  // User I/Os
  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 analyzer3_data = 16'h0;
  assign analyzer3_clock = 1'b1;
 // assign analyzer4_data = 16'h0;
  assign analyzer4_clock = 1'b1;

  //////////////////////////////////////////////////////////////////////
  //
  // VGA Output
  //
  //////////////////////////////////////////////////////////////////////
  wire [1:0] color_sel;
  wire [4:0] note_now;
  wire start_video, done_video;

  write vga_test (switch[7], clock_27mhz, 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, pixel_clock, note_now, start_video,
done_video, color_sel);

  //////////////////////////////////////////////////////////////////////////
  //
  // BRAIN
  //
  //////////////////////////////////////////////////////////////////////////

  brain brain_instantiation(
  .clk(pixel_clock),
  .reset(switch[7]),
  .done(done_video),
  .note_in(value[4:0]),
  .note_now(note_now),
  .start(start_video)
  );

  sync sync_inst(pixel_clock, switch[6:5], color_sel);

endmodule

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

Pitch Detection Code (Yun Wu)

```
**Note: audio module originally developed by Nathan Ickes, has been modified for this
project
module audio (reset, clock_27mhz, audio_reset_b, ac97_sdata_out, ac97_sdata_in,
                ac97_synch, ac97_bit_clock, mode, volume, source, left_in_data,
                     fftdone, rdy, busy, ovflo, xn_index, xk_index, xk_re, xk_im, xn_re,
xn_im,
                     n_fft, n_fft_we, fwd_inv, fwd_inv_we, scale_sch, scale_sch_we, start, ce,
                     waddress, resetout, value, max_index, rom_enable);

  input reset, clock_27mhz;
  output audio_reset_b;
  output ac97_sdata_out;
  input ac97_sdata_in;
  output ac97_synch;
  input ac97_bit_clock;
  input [1:0] mode;
  input [4:0] volume;
  input source;

  //**********in order to look at output of A/D on logic analyzer
  output [19:0] left_in_data;

  wire ready;
  wire [7:0] command_address;
  wire [15:0] command_data;
  wire command_valid;
  reg [19:0] left_out_data, right_out_data;
  wire [19:0] left_in_data, right_in_data, sine_data, square_data;

  //
  // Reset controller
  //

  reg audio_reset_b;
  reg [9:0] reset_count;
```

```
  always @(posedge clock_27mhz) begin
    if (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

  ac97 ac97(ready, command_address, command_data, command_valid,
            left_out_data, 1'b1, right_out_data, 1'b1, left_in_data,
            right_in_data, ac97_sdata_out, ac97_sdata_in, ac97_synch,
            ac97_bit_clock);

  ac97commands cmds(clock_27mhz, ready, command_address, command_data,
                    command_valid, volume, source);

  sinewave sine(clock_27mhz, ready, sine_data);

  squarewave square(clock_27mhz, ready, square_data);

 ///*****************************************///
 //My code added here

  //*******feature 2'd2
  wire [14:0] waddr, raddr;
  wire [15:0] left_out_data_temp;
  wire wea;
  //instantiate a ram and ramcontroller for testing  ---feature 2'd2
  ram_dptest big_ram (waddr, raddr, ac97_bit_clock, ac97_bit_clock, left_in_data[19:4],
left_out_data_temp, wea);

  ram_controller ram_controller1(reset, ac97_bit_clock, ready, wea, waddr, raddr);


 //********fft stuff
  output fftdone, rdy, busy, ovflo;  //debugging
  output [9:0] xn_index, xk_index;
  output [15:0] xk_re, xk_im;
  output [15:0] xn_re, xn_im;
  output start;
  output [10:0] waddress;
```

```
  output resetout;
  output n_fft_we, fwd_inv_we;
  output [4:0] value;
  output fwd_inv, ce, scale_sch_we;
  output [9:0] scale_sch, max_index;
  output [4:0] n_fft;
  output rom_enable;

  wire [10:0] waddress, raddress;
  wire wrena;
  wire [4:0] n_fft;
  wire n_fft_we, fwd_inv, fwd_inv_we, scale_sch_we, start, ce;
  wire [9:0] scale_sch;

  wire fftdone, rdy, busy, edone, done, ovflo;
  wire [15:0] xk_re, xk_im, xn_re, xn_im;
  wire [9:0] xn_index, xk_index, max_index;

  wire [15:0] right_data_to_fft;
  wire resetout;
  wire rfd; //high during load operation
  wire rom_enable;

  reg resetswitch;
 // assign xn_re = {xn_index[3:0],12'h000};
  assign resetout=resetswitch;

  newfftcontroller fftcontrol1(ac97_bit_clock, resetswitch, waddress[10], n_fft, n_fft_we,
fwd_inv, fwd_inv_we, scale_sch, scale_sch_we, start);

 newfft1024 newfft1 (xn_re, 16'h0000, start, edone, nfft,      nfft_we, fwd_inv,
fwd_inv_we,   scale_sch, scale_sch_we,       resetswitch, ac97_bit_clock,
                                    xk_re, xk_im, xn_index, xk_index,    rfd,      busy, rdy,
edone, fftdone, ovflo);

//new artificial fft
/* newfft1024 newfft1 ({xk_index[3:0], 12'h000}, 16'h0000, start, edone, nfft,
          nfft_we, fwd_inv, fwd_inv_we,         scale_sch, scale_sch_we,       resetswitch,
ac97_bit_clock,
                                    xk_re, xk_im, xn_index, xk_index,    rfd,      busy, rdy,
edone, fftdone, ovflo);

                  */


//artificial fft
```

```
/*    xfft1024_v1_1 fft(.xn_re({xk_index[3:0], 12'h000}), .xn_im(16'h0000),
.reset(resetswitch), .start(start), .mrd(fftdone), .n_fft(n_fft), .n_fft_we(n_fft_we),
.fwd_inv(fwd_inv), .fwd_inv_we(fwd_inv_we),
                                    .scale_sch(scale_sch), .scale_sch_we(scale_sch_we),
.clk(ac97_bit_clock),
                                    .ce(ce), .xk_re(xk_re), .xk_im(xk_im),
.xn_index(xn_index), .xk_index(xk_index),
                                    .rdy(rdy), .busy(busy), .edone(edone), .done(fftdone),
.ovflo(ovflo));              */




  //real fft
  /* xfft1024_v1_1 fft(.xn_re(xn_re), .xn_im(16'h0000), .reset(resetswitch), .start(start),
.mrd(fftdone), .n_fft(n_fft), .n_fft_we(n_fft_we), .fwd_inv(fwd_inv),
.fwd_inv_we(fwd_inv_we),
                                    .scale_sch(scale_sch), .scale_sch_we(scale_sch_we),
.clk(ac97_bit_clock),
                                    .ce(ce), .xk_re(xk_re), .xk_im(xk_im),
.xn_index(xn_index), .xk_index(xk_index),
                                    .rdy(rdy), .busy(busy), .edone(edone), .done(fftdone),
.ovflo(ovflo)); */

  //real ram (storing the input audio data)
   ram_dp two_k_ram(waddress, {~waddress[10], xn_index[9:0]}, ac97_bit_clock,
ac97_bit_clock, right_in_data[19:4] , xn_re, wrena);

    //artificial waddress to the input data
  // ram_dp two_k_ram(waddress, {~waddress[10], xk_index[9:0]}, ac97_bit_clock,
ac97_bit_clock, {waddress[10:0], 5'b00000} , xn_re, wrena);

   ram_controller_fft fft_ram_controller1(resetswitch, ac97_bit_clock, ready, wrena,
waddress, raddress);


  afterfft afterfft1(xk_index, xk_im, xk_re, ac97_bit_clock, reset, value, max_index,
rom_enable);

   always @(mode or left_in_data or right_in_data or sine_data or square_data or
left_out_data_temp or resetout)
     case (mode)
       2'd0: begin
             //audio playback
             left_out_data = left_in_data;
             right_out_data = right_in_data;
```

```
            resetswitch=0;
        end
        2'd1: begin
            resetswitch=1;
            //used for resetting


            //left_out_data = 20'h00000;
            //right_out_data = 20'h00000;
        end
        2'd2: begin
            //audio playback with echo effect!!!
            left_out_data={left_out_data_temp, 4'b0000};
            //left_out_data = sine_data;
            //right_out_data = sine_data;
        end
        2'd3: begin
           //  right_out_data={right_data_to_fft, 4'b0000};
            //left_out_data = square_data;
            //right_out_data = square_data;
        end
      endcase

endmodule

module beeper (reset, clock_27mhz, beep, enable);

  input reset, clock_27mhz, enable;
  output beep;

  reg [15:0] count;
  reg clock_1khz;

  always @(posedge clock_27mhz)
    if (reset)
      begin
            count <= 0;
            clock_1khz <= 0;
      end
    else if (count == 13499)
      begin
            clock_1khz <= ~clock_1khz;
            count <= 0;
     end
    else
      count <= count+1;
```

```
  assign beep = enable && clock_1khz;

endmodule

module volume (reset, clock, up, down, vol, disp);

  input reset, clock, up, down;
  output [3:0] vol;
  output [39:0] disp;

  reg [3:0] vol;
  reg [39:0] disp;
  reg old_up, old_down;

  always @(posedge clock)
    if (reset)
      begin
            vol <= 0;
            old_up <= 0;
            old_down <= 0;
      end
    else
      begin
            if ((up == 1) && (old_up == 0) && (vol < 15))
              vol <= vol+1;
            else if ((down == 1) && (old_down == 0) && (vol > 0))
              vol <= vol-1;
            old_up <= up;
            old_down <= down;
      end

  always @(vol)
    case (vol[3:1])
      0: disp <= { 5{8'b00000000}};
      1: disp <= { 5{8'b01000000}};
      2: disp <= { 5{8'b01100000}};
      3: disp <= { 5{8'b01110000}};
      4: disp <= { 5{8'b01111000}};
      5: disp <= { 5{8'b01111100}};
      6: disp <= { 5{8'b01111110}};
      7: disp <= { 5{8'b01111111}};
    endcase

endmodule
```

```
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

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

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;

  ////SET GAIN
          4'hB: //Set mic input gain
            command <=24'h0E_8048;
          //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


module sinewave (clock, ready, pcm_data);

    input clock;
    input ready;
    output [19:0] pcm_data;

    reg rdy, old_ready;
    reg [8:0] index;
    reg [19:0] pcm_data;

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

```
   index <= 8'h00;
   // synthesis attribute init of index is "00";
   pcm_data <= 20'h00000;
   // synthesis attribute init of pcm_data is "00000";
end

always @(posedge clock) begin
   if (rdy && ~old_ready)
        index <= index+1;
   old_ready <= rdy;
   rdy <= ready;
end

always @(index) begin
   case (index[5:0])
     6'h00: pcm_data <= 20'h00000;
     6'h01: pcm_data <= 20'h0C8BD;
     6'h02: pcm_data <= 20'h18F8B;
     6'h03: pcm_data <= 20'h25280;
     6'h04: pcm_data <= 20'h30FBC;
     6'h05: pcm_data <= 20'h3C56B;
     6'h06: pcm_data <= 20'h471CE;
     6'h07: pcm_data <= 20'h5133C;
     6'h08: pcm_data <= 20'h5A827;
     6'h09: pcm_data <= 20'h62F20;
     6'h0A: pcm_data <= 20'h6A6D9;
     6'h0B: pcm_data <= 20'h70E2C;
     6'h0C: pcm_data <= 20'h7641A;
     6'h0D: pcm_data <= 20'h7A7D0;
     6'h0E: pcm_data <= 20'h7D8A5;
     6'h0F: pcm_data <= 20'h7F623;
     6'h10: pcm_data <= 20'h7FFFF;
     6'h11: pcm_data <= 20'h7F623;
     6'h12: pcm_data <= 20'h7D8A5;
     6'h13: pcm_data <= 20'h7A7D0;
     6'h14: pcm_data <= 20'h7641A;
     6'h15: pcm_data <= 20'h70E2C;
     6'h16: pcm_data <= 20'h6A6D9;
     6'h17: pcm_data <= 20'h62F20;
     6'h18: pcm_data <= 20'h5A827;
     6'h19: pcm_data <= 20'h5133C;
     6'h1A: pcm_data <= 20'h471CE;
     6'h1B: pcm_data <= 20'h3C56B;
     6'h1C: pcm_data <= 20'h30FBC;
     6'h1D: pcm_data <= 20'h25280;
     6'h1E: pcm_data <= 20'h18F8B;
```

```
      6'h1F: pcm_data <= 20'h0C8BD;
      6'h20: pcm_data <= 20'h00000;
      6'h21: pcm_data <= 20'hF3743;
      6'h22: pcm_data <= 20'hE7075;
      6'h23: pcm_data <= 20'hDAD80;
      6'h24: pcm_data <= 20'hCF044;
      6'h25: pcm_data <= 20'hC3A95;
      6'h26: pcm_data <= 20'hB8E32;
      6'h27: pcm_data <= 20'hAECC4;
      6'h28: pcm_data <= 20'hA57D9;
      6'h29: pcm_data <= 20'h9D0E0;
      6'h2A: pcm_data <= 20'h95927;
      6'h2B: pcm_data <= 20'h8F1D4;
      6'h2C: pcm_data <= 20'h89BE6;
      6'h2D: pcm_data <= 20'h85830;
      6'h2E: pcm_data <= 20'h8275B;
      6'h2F: pcm_data <= 20'h809DD;
      6'h30: pcm_data <= 20'h80000;
      6'h31: pcm_data <= 20'h809DD;
      6'h32: pcm_data <= 20'h8275B;
      6'h33: pcm_data <= 20'h85830;
      6'h34: pcm_data <= 20'h89BE6;
      6'h35: pcm_data <= 20'h8F1D4;
      6'h36: pcm_data <= 20'h95927;
      6'h37: pcm_data <= 20'h9D0E0;
      6'h38: pcm_data <= 20'hA57D9;
      6'h39: pcm_data <= 20'hAECC4;
      6'h3A: pcm_data <= 20'hB8E32;
      6'h3B: pcm_data <= 20'hC3A95;
      6'h3C: pcm_data <= 20'hCF044;
      6'h3D: pcm_data <= 20'hDAD80;
      6'h3E: pcm_data <= 20'hE7075;
      6'h3F: pcm_data <= 20'hF3743;
    endcase // case(index[8:2])
  end // always @ (index)

endmodule



module squarewave (clock, ready, pcm_data);

  input clock;
  input ready;
  output [19:0] pcm_data;
```

```
  reg old_ready;
  reg [6:0] index;
  reg [19:0] pcm_data;

  initial begin
    old_ready <= 1'b0;
    // synthesis attribute init of old_ready is "0";
    index <= 7'h00;
    // synthesis attribute init of index is "00";
    pcm_data <= 20'h00000;
    // synthesis attribute init of pcm_data is "00000";
  end

  always @(posedge clock) begin
    if (ready && ~old_ready)
        index <= index+1;
    old_ready <= ready;
  end

  always @(index) begin
    if (index[6])
        pcm_data <= 20'hF0F00;
    else
        pcm_data <= 20'h05555;
  end // always @ (index)

endmodule

//////////////////////////////////////////////////////////////////////////////
//
// 6.111 FPGA Labkit -- Alphanumeric Display Interface
//
//
// Created: November 5, 2003
// Author: Nathan Ickes
//
//////////////////////////////////////////////////////////////////////////////

module display (reset, clock_27mhz,
                disp_blank, disp_clock, disp_rs, disp_ce_b,
                disp_reset_b, disp_data_out, dots);

  input  reset, clock_27mhz;
  output disp_blank, disp_clock, disp_data_out, disp_rs, disp_ce_b,
         disp_reset_b;
  input [639:0] dots;
```

```
reg disp_data_out, disp_rs, disp_ce_b, disp_reset_b;

////////////////////////////////////////////////////////////////////////
//
// Display Clock
//
// Generate a 500kHz clock for driving the displays.
//
////////////////////////////////////////////////////////////////////////

reg [4:0] count;
reg [7:0] reset_count;
reg clock;
wire dreset;

always @(posedge clock_27mhz)
  begin
        if (reset)
          begin
            count = 0;
            clock = 0;
          end
        else if (count == 26)
          begin
            clock = ~clock;
            count = 5'h00;
          end
        else
          count = count+1;
  end

always @(posedge clock_27mhz)
  if (reset)
    reset_count <= 100;
  else
    reset_count <= (reset_count==0) ? 0 : reset_count-1;

assign dreset = (reset_count != 0);

assign disp_clock = ~clock;

////////////////////////////////////////////////////////////////////////
//
// Display State Machine
//
```

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

reg [7:0] state;
reg [9:0] dot_index;
reg [31:0] control;

assign disp_blank = 1'b0; // low <= not blanked

always @(posedge clock)
  if (dreset)
    begin
          state <= 0;
          dot_index <= 0;
          control <= 32'h7F7F7F7F;
    end
  else
    casex (state)
          8'h00:
            begin
              // Reset displays
              disp_data_out <= 1'b0;
              disp_rs <= 1'b0; // dot register
              disp_ce_b <= 1'b1;
              disp_reset_b <= 1'b0;
              dot_index <= 0;
              state <= state+1;
            end

          8'h01:
            begin
              // End reset
              disp_reset_b <= 1'b1;
              state <= state+1;
            end

          8'h02:
            begin
              // Initialize dot register
              disp_ce_b <= 1'b0;
              disp_data_out <= 1'b0; // dot_index[0];
              if (dot_index == 639)
                    state <= state+1;
              else
                    dot_index <= dot_index+1;
            end
```

```
            8'h03:
              begin
                // Latch dot data
                disp_ce_b <= 1'b1;
                dot_index <= 31;
                state <= state+1;
              end

            8'h04:
              begin
                // Setup the control register
                disp_rs <= 1'b1; // Select the control register
                disp_ce_b <= 1'b0;
                disp_data_out <= control[31];
                control <= {control[30:0], 1'b0};
                if (dot_index == 0)
                     state <= state+1;
                else
                     dot_index <= dot_index-1;
              end

            8'h05:
              begin
                // Latch the control register data
                disp_ce_b <= 1'b1;
                dot_index <= 639;
                state <= state+1;
              end

            8'h06:
              begin
                // Load the user's dot data into the dot register
                disp_rs <= 1'b0; // Select the dot register
                disp_ce_b <= 1'b0;
                disp_data_out <= dots[dot_index];
                if (dot_index == 0)
                     state <= 5;
                else
                     dot_index <= dot_index-1;
              end
        endcase

endmodule

module afterfft(xn_index,xk_im,xk_re,clk,reset, value, max_index, rom_enable);
//module that encapsulates everything after the fft
```

```
    input [9:0] xn_index;
    input [15:0] xk_im;
    input [15:0] xk_re;
    input clk;
    input reset;
    output [4:0] value;
    output [9:0] max_index;
    output rom_enable;

    wire [31:0] xk_re_sq, xk_im_sq;
    wire [32:0] xk_mag_sqd;
    wire [9:0] rd_addr;
    wire [9:0] max_index;
    wire [4:0] value_int;
    wire rom_enable;

        fftmagnitude fftmag1(xn_index, xk_im, xk_re, clk, reset, xk_mag_sqd, xk_re_sq,
xk_im_sq, rd_addr);
        comparator compare1( xk_mag_sqd, rd_addr, clk, reset, max_index, rom_enable);
        notesrom rom1(.addr(max_index), .clk(clk), .dout(value_int), .en(rom_enable));
        valuereg valuereg1(clk,reset,value_int, rom_enable, value);
endmodule


module comparator(rd_data,rd_addr, clk, reset, max_index, rom_enable);
    input [32:0] rd_data;
    input [9:0] rd_addr;
    input clk, reset;
    output [9:0] max_index;
    output rom_enable;

    reg [9:0] max_index, max_index_int;
    reg [32:0] max_val, max_val_int;
    reg rom_enable;

    always @ (posedge clk) begin
        if (reset) begin
                max_val_int<=0;
                max_index_int<=0;
                rom_enable<=0;
            end

            else if (rd_addr == 10'b1111111111) begin
                max_val_int<=0;
                max_index_int<=0;
```

```
                rom_enable<=1;
            end

            else if (rd_data > max_val_int) begin
              max_val_int <= rd_data;
              max_index_int <=rd_addr;
              rom_enable<=0;

            end

            else
              rom_enable<=0;




                max_val<= max_val_int;
                max_index<=max_index_int;

  end

endmodule

module fftmagnitude(xn_index,xk_im,xk_re,clk,reset, xk_mag_sqd, xk_re_sq,
xk_im_sq, rd_addr);
  input [9:0] xn_index;
  input [15:0] xk_im;
  input [15:0] xk_re;
  input clk;
  input reset;
  output [31:0] xk_re_sq, xk_im_sq;
  output [32:0] xk_mag_sqd;
  output [9:0] rd_addr;


  wire [31:0] xk_re_sq, xk_im_sq;
  wire [32:0] xk_mag_sqd;
  reg [9:0] rd_addr, rd_addr_int;

  multa sq_re(.clk(clk), .a(xk_re), .b(xk_re), .q(xk_re_sq));
  multa sq_im(.clk(clk), .a(xk_im), .b(xk_im), .q(xk_im_sq));
  adder adder1(.A(xk_re_sq), .B(xk_im_sq), .C_IN(0), .Q(xk_mag_sqd), .CLK(clk));

  always @ (posedge clk) begin
            if (reset) begin
                rd_addr<=0;
```

```
                end

                else begin
                rd_addr_int <= xn_index;
                rd_addr <= rd_addr_int;
                end

    end

endmodule

module valuereg(clk,reset,value_int, rom_enable, value);
    input clk;
    input reset;
    input rom_enable;
    input [4:0] value_int;
    output [4:0] value;

    reg [4:0] value;
    reg rom_enable_d;

    always @ (posedge clk) begin
            if (reset)
                    value<=5'b00000;
            if (rom_enable_d)
                    value<=value_int;

            rom_enable_d<=rom_enable;


    end

endmodule


module newfftcontroller (clk, reset, ram_waddr_msb, n_fft, n_fft_we, fwd_inv,
fwd_inv_we, scale_sch, scale_sch_we, start);

/* newfft1024
(xn_re,xn_im,start,unload,nfft,nfft_we,fwd_inv,fwd_inv_we,scale_sch,scale_sch_we,sclr
,clk,xk_re,
        xk_im,xn_index,xk_index,rfd,busy,dv,edone,done,ovflo);      */

input clk, reset;
input ram_waddr_msb;  //from ram controller, so we know which ram "block" of 1024
(top or bottom) is getting written
```

```
//input [15:0] xn_re;
//input [15:0] xn_im;
//input reset;
output start;
//input mrd;     tie the DONE signal of fft module back to mrd for automatic data unloading
output [4:0] n_fft;
output n_fft_we;
output fwd_inv;
output fwd_inv_we;
output [9:0] scale_sch;
output scale_sch_we;
//input clk;
//output ce;
//output [3:0] state;  //state is outputted for debugging

reg start, n_fft_we, fwd_inv_we, scale_sch_we;
reg start_d, start_d2, n_fft_we_d, fwd_inv_we_d, scale_sch_we_d;

//ram_waddr_msb transition tells us data has been written in a 1024 block in ram, it's ok to start fft
reg prev_ram_waddr_msb;

//internal
reg [1:0] state;
reg [1:0] next;

wire [4:0] n_fft;
wire [9:0] scale_sch;
wire ce;
wire fwd_inv;

assign n_fft = 5'b01010;
assign fwd_inv =1'b1;
//assign ce = 1'b1;

//use most conservative scaling schedule to avoid overflows
assign scale_sch = 10'b1010101110;

parameter IDLE = 0;
parameter setDefaults = 1;
parameter pulseStart = 2;
parameter waitforRAMcontroller = 3;  //wait for RAM controller's waddress MSB to go from 1 to 0 or 0 to 1
```

```
always @ (posedge clk)
	begin
		if (reset) state<=IDLE;
		else state<=next;

	start<=start_d;
	start_d<=start_d2;
	n_fft_we<=n_fft_we_d;
	fwd_inv_we<=fwd_inv_we_d;
	scale_sch_we<=scale_sch_we_d;
	prev_ram_waddr_msb<=ram_waddr_msb;
	end

always @ (state or ram_waddr_msb or prev_ram_waddr_msb) begin
	//defaults:
	start_d2=0;
	n_fft_we_d=0;
	fwd_inv_we_d=0;
	scale_sch_we_d=0;

	case (state)
		IDLE: next=setDefaults;

		setDefaults:
			begin
				n_fft_we_d=1;
				fwd_inv_we_d=1;
				scale_sch_we_d=1;
				next = pulseStart;
			end

		pulseStart:
			begin
				start_d2=1;
				next=waitforRAMcontroller;
			end

		waitforRAMcontroller:
			begin
			  if (ram_waddr_msb == prev_ram_waddr_msb)
					next=waitforRAMcontroller;
			  else
				next=pulseStart;
			end
```

```
                default: next = IDLE;

        endcase
end


endmodule


module ram_controller_fft(reset, clk, ready, wea, waddr, raddr);
        input reset, clk, ready;
        output wea; //wea_i;
        output [10:0] waddr, raddr;
            //output oldready;

        reg wea;
            reg wea_i;
        reg oldready;
        reg [10:0] waddr, waddr_iii, waddr_ii, waddr_i, raddr;
        reg [3:0] ready_count;  //when ready_count ==4'b1111, then sample (gets us a
sampling rate of 3kHz)

        initial begin
                oldready<= 1'b0;
        end

        always @ (posedge clk) begin
            if (reset)
                                begin
                                  wea_i<=1'b0;
                                  waddr_i<=1'b0;
                                  raddr<=1'b1;
                                  ready_count<=4'b0000;
                                end

                        /*if (ready && (!oldready)) begin
                                wea_i<=1'b1;
                                waddr_i<=waddr_i+1;  end
                        else
                                wea_i<=1'b0; */
            else begin
            if (ready && (!oldready)) begin
                        ready_count<=ready_count+1;
            end

            if (ready_count == 4'b1111 && (!oldready) && ready) begin
```

```
                wea_i<=1'b1;
                waddr_i<=waddr_i+1;
        end


        else
                        wea_i<=1'b0;
        end


                oldready<=ready;
                wea<=wea_i;
                waddr_ii<=waddr_i;
                waddr_iii<=waddr_ii;
                waddr<=waddr_iii;
                raddr<=waddr_iii+1;
        end

endmodule
```

Video Display Code (Shi Ling Seow)

```
module brain (clk, reset, done, note_in, note_now, start);

        input clk, reset, done;
        input [4:0] note_in;
        output start;
        output [4:0] note_now;
        reg start, start_int;
        reg [1:0] state, next;
        reg [4:0] note_temp, note_now;

        parameter IDLE = 0;
        parameter RESTART = 1;
        parameter DECIDE = 2;
        parameter WAIT = 3;

        always @ (posedge clk or posedge reset) begin
                if (reset) begin
                        start <= 0;
                        state <= IDLE;
                        note_temp <= note_in;
                end
                else if (state == 1) begin
                        start <= start_int;
```

```
                state <= next;
                note_temp <= note_in;
        end
        else begin
                start <= start_int;
                state <= next;
                note_temp <= note_temp;
        end
end

always @ * begin
        case (state)

        IDLE: begin
                        start_int <= 0;
                        note_now <= note_temp;
                        next <= RESTART;
                 end
        RESTART: begin
                                start_int <= 0;
                                note_now <= note_temp;
                                next <= DECIDE;
                         end
        DECIDE: begin
                        if (note_temp != note_in) begin
                                start_int <= 1;
                                note_now <= note_temp;
                                next <= WAIT;
                        end
                        else begin
                                start_int <= 0;
                                note_now <= note_temp;
                                next <= DECIDE;
                        end
                  end
        WAIT:  begin
                        if (done) begin
                                start_int <= 0;
                                note_now <= note_temp;
                                next <= RESTART;
                        end
                        else begin
                                start_int <= 1;
                                note_now <= note_temp;
                                next <= WAIT;
                        end
```

```
                        end
                endcase
        end
endmodule

module sync (clk, in, in_sync);

        input clk;
        input [1:0] in;
        output [1:0] in_sync;
        reg [1:0] in_sync1, in_sync2;
        wire [1:0] in_sync;

        assign in_sync = in_sync2;

        always @ (posedge clk) begin
                in_sync1 <= in;
                in_sync2 <= in_sync1;
        end

endmodule
```

Note: Modified from Nathan Ickes's vga.v code from the Labkit Documentation

```
module write (reset, clock_27mhz, 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, pixel_clock, note_now, start, done,
             color_sel);

  input [1:0] color_sel;
  input start;
  input [4:0] note_now;
  output pixel_clock;
  input reset; // Active high reset, synchronous with 27MHz clock
  input clock_27mhz; // 27MHz input clock
  output done;
  output [7:0] vga_out_red, vga_out_green, vga_out_blue; // Outputs to DAC
  output vga_out_sync_b, vga_out_blank_b; // Composite sync/blank outputs to DAC
  output vga_out_pixel_clock; // Pixel clock for DAC
  output vga_out_hsync, vga_out_vsync; // Sync outputs to VGA connector

  ////////////////////////////////////////////////////////////////////////
  //
  // Timing values
  //
  ////////////////////////////////////////////////////////////////////////
```

```
  // 1024 X 768 @ 62Hz with a 65MHz pixel clock
`define H_ACTIVE        1024 // pixels
`define H_FRONT_PORCH   16 // pixels
`define H_SYNCH         96 // pixels
`define H_BACK_PORCH    176 // pixels
`define H_TOTAL         1312 // pixels

`define V_ACTIVE      768 // lines
`define V_FRONT_PORCH  1 // lines
`define V_SYNCH        3 // lines
`define V_BACK_PORCH   28 // lines
`define V_TOTAL       800 // lines

  ////////////////////////////////////////////////////////////////////////
  //
  // Internal signals
  //
  ////////////////////////////////////////////////////////////////////////

  wire pixel_clock;
  reg prst, pixel_reset; // Active high reset, synchronous with pixel clock

  reg [7:0] vga_out_red, vga_out_blue, vga_out_green;
  wire vga_out_sync_b, vga_out_blank_b;
  reg hsync1, hsync2, vga_out_hsync, vsync1, vsync2, vga_out_vsync;

  reg [10:0] pixel_count; // Counts pixels in each line
  reg [10:0] line_count; // Counts lines in each frame

  ////////////////////////////////////////////////////////////////////////
  //
  // Generate the pixel clock (78.750MHz)
  //
  ////////////////////////////////////////////////////////////////////////

  // synthesis attribute period of clock_27mhz is 37ns;


  DCM vga_dcm (.CLKIN(clock_27mhz),
                .RST(1'b0),
                .CLKFX(pixel_clock));

defparam vga_dcm.CLKDV_DIVIDE = 2.0;
defparam vga_dcm.CLKFX_DIVIDE = 5;
defparam vga_dcm.CLKFX_MULTIPLY = 12;
```

```
defparam vga_dcm.CLKIN_DIVIDE_BY_2 = "FALSE";
defparam vga_dcm.CLKIN_PERIOD = 37.04;                          // non-simulatable
defparam vga_dcm.CLKOUT_PHASE_SHIFT = "NONE";
defparam vga_dcm.CLK_FEEDBACK = "1X";
defparam vga_dcm.DESKEW_ADJUST = "SYSTEM_SYNCHRONOUS";          // non-
simulatable
defparam vga_dcm.DFS_FREQUENCY_MODE = "LOW";
defparam vga_dcm.DLL_FREQUENCY_MODE = "LOW";
defparam vga_dcm.PHASE_SHIFT = 0;
defparam vga_dcm.STARTUP_WAIT = "TRUE";             // non-simulatable

   assign vga_out_pixel_clock = ~pixel_clock;

  always @(posedge pixel_clock)
    begin
          prst <= reset;
          pixel_reset <= prst;
    end

  ////////////////////////////////////////////////////////////////////////
  //
  // Pixel and Line Counters
  //
  ////////////////////////////////////////////////////////////////////////

  always @(posedge pixel_clock)
    if (pixel_reset)
      begin
            pixel_count <= 0;
            line_count <= 0;
      end
    else if (pixel_count == (`H_TOTAL-1)) // last pixel in the line
      begin
            pixel_count <= 0;
            if (line_count == (`V_TOTAL-1)) // last line of the frame
              line_count <= 0;
            else
              line_count <= line_count + 1;
      end
    else
      pixel_count <= pixel_count +1;

  ////////////////////////////////////////////////////////////////////////
  //
  // Sync and Blank Signals
  //
```

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

  always @ (posedge pixel_clock)
    begin
          if (pixel_reset)
            begin
              hsync1 <= 1;
              hsync2 <= 1;
              vga_out_hsync <= 1;
              vsync1 <= 1;
              vsync2 <= 1;
              vga_out_vsync <= 1;
            end
          else
            begin

              // Horizontal sync
              if (pixel_count == (`H_ACTIVE+`H_FRONT_PORCH))
                hsync1 <= 0; // start of h_sync
              else if (pixel_count == (`H_ACTIVE+`H_FRONT_PORCH+`H_SYNCH))
                hsync1 <= 1; // end of h_sync

              // Vertical sync
              if (pixel_count == (`H_TOTAL-1))
                begin
                      if (line_count == (`V_ACTIVE+`V_FRONT_PORCH))
                        vsync1 <= 0; // start of v_sync
                      else if (line_count ==
(`V_ACTIVE+`V_FRONT_PORCH+`V_SYNCH))
                        vsync1 <= 1; // end of v_sync
                end

            end

          // Delay hsync and vsync by two cycles to compensate for 2 cycles of
          // pipeline delay in the DAC.
          hsync2 <= hsync1;
          vga_out_hsync <= hsync2;
          vsync2 <= vsync1;
          vga_out_vsync <= vsync2;

    end

  // Blanking
  assign vga_out_blank_b = ((pixel_count<`H_ACTIVE) & (line_count<`V_ACTIVE));
          //high during non-blanking
```

```
  // Composite sync
  assign vga_out_sync_b = hsync1 ^ vsync1;

  //////////////////////////////////////////////////////////////////////
  //
  // Output From RAM
  //
  //////////////////////////////////////////////////////////////////////
        `define ROM_SIZE 47104
        `define BLOCK_SIZE 2048
        reg done;
        reg done_int;
        reg [4:0] note_sel;
        reg we, we_int;
        reg [3:0] state, next;
        reg [19:0] ram_add;
        reg [15:0] rom_add;
        reg [5:0] block_count;
        reg [20:0] rom_count, write_count, read_count, width_count, length_count,
pointer;
    wire ram_in;

  // RAM control signals

  rom rom_instantiation (.clk(pixel_clock), .addr(rom_add), .dout(ram_in));
  ram ram_instantiation (.clk(pixel_clock), .addr(ram_add), .din(ram_in), .dout(ram_out),
.we(we));

  // RAM control signals
  parameter IDLE = 0;
  parameter SET = 1;
  parameter SETRAM = 2;
  parameter SETROM = 3;
  parameter WRITE = 4;
  parameter WAIT = 5;
  parameter READ = 6;
  parameter READWAIT = 7;
  parameter TREBLERAM = 8;
  parameter TREBLEROM = 9;
  parameter TREBLEWRITE = 10;
  parameter WRITERAM = 11;
  parameter WRITEROM = 12;
  parameter WRITING = 13;

  always @ (posedge pixel_clock)
```

```
	begin
	we <= we_int;
	done <= done_int;
	//if (note_sel == 22) note_sel <= 0;
	//else note_sel <= note_sel + 1;

	if (reset) begin
		note_sel <= 21;
		block_count <= 0;
		width_count <= 0;
		state <= IDLE;
		rom_count <= 0;
		write_count <= 0;
		length_count <= 0;
		pointer <= 0;
		read_count <= 0;
	end
	else if (state == 1) begin
		block_count <= 0;
		width_count <= 0;
		write_count <= 0;
		read_count <= 0;
		rom_count <= 21 * 2048;	// blank bar
		length_count <= 0;
		pointer <= pointer;
		state <= next;
end
else if (state == 2) begin
		block_count <= 0;
		width_count <= 0;
		write_count <= pointer;
		rom_count <= rom_count + 1;
		length_count <= 0;
		pointer <= pointer;
		state <= next;
	end
	else if (state == 11 || state == 8) begin
		block_count <= block_count;
		width_count <= 0;
		write_count <= pointer;
		rom_count <= rom_count + 1;
		length_count <= 0;
		pointer <= pointer;
		state <= next;
	end
	else if (state == 3) begin
```

```
                block_count <= 0;
                width_count <= width_count + 1;
                write_count <= write_count + 1;
                rom_count <= rom_count + 1;
                length_count <= 0;
                pointer <= pointer;
                state <= next;
        end
        else if (state == 12 || state == 9) begin
                block_count <= block_count;
                width_count <= width_count + 1;
                write_count <= write_count + 1;
                rom_count <= rom_count + 1;
                length_count <= 0;
                pointer <= pointer;
                state <= next;
        end
        else if (state == 4) begin
                if ((length_count == 63) && (width_count == 30))begin
                        width_count <= width_count + 1;
                        length_count <= length_count;
                        block_count <= block_count;
                        if (rom_count == (21 * 2048) + 2047) rom_count <= 21 * 2048;
                        else rom_count <= rom_count + 1;
                        write_count <= write_count + 1;
                        if (block_count == 31) begin
                                if (pointer == ((`H_ACTIVE * `V_ACTIVE) -
(`H_ACTIVE * 63) - 32)) pointer <= 0;
                                else pointer <= write_count + 2;
                        end
                        else pointer <= pointer + 32;
                end
                else if ((length_count == 63) && (width_count == 31)) begin
                        width_count <= 0;
                        length_count <= 0;
                        if (block_count == 31) block_count <= 0;
                        else block_count <= block_count + 1;
                        if (rom_count == (21 * 2048) + 2047) rom_count <= 21 * 2048;
                        else rom_count <= rom_count + 1;
                        write_count <= pointer;
                        pointer <= pointer;
                        state <= next;
                end
                else if (width_count == 31) begin
                        width_count <= 0;
                        block_count <= block_count;
```

```
                        length_count <= length_count + 1;
                        if (rom_count == (21 * 2048) + 2047) rom_count <= 21 * 2048;
                        else rom_count <= rom_count + 1;
                        write_count <= write_count + 1024 - 31;
                        pointer <= pointer;
                        state <= next;
                end
                else begin
                        block_count <= block_count;
                        width_count <= width_count + 1;
                        write_count <= write_count + 1;
                        if (rom_count == (21 * 2048) + 2047) rom_count <= 21 * 2048;
                        else rom_count <= rom_count + 1;
                        length_count <= length_count;
                        pointer <= pointer;
                        state <= next;
                end
        end
        else if (state == 13 || state == 10) begin
                if ((length_count == 63) && (width_count == 30))begin
                        width_count <= width_count + 1;
                        length_count <= length_count;
                        block_count <= block_count;
                        if (rom_count == (20 * 2048) + 2047) rom_count <= note_now *
2048;
                        else if (rom_count == (note_now * 2048) + 2047) rom_count <=
note_now * 2048;
                        else rom_count <= rom_count + 1;
                        write_count <= write_count + 1;
                        if (block_count == 31) begin
                                if (pointer == ((`H_ACTIVE * `V_ACTIVE) -
(`H_ACTIVE * 63) - 32)) pointer <= 0;
                                else pointer <= write_count + 2;
                        end
                        else pointer <= pointer + 32;
                end
                else if ((length_count == 63) && (width_count == 31)) begin
                        width_count <= 0;
                        length_count <= 0;
                        if (block_count == 31) block_count <= 0;
                        else block_count <= block_count + 1;
                        if (rom_count == (20 * 2048) + 2047) rom_count <= note_now *
2048;
                        else if (rom_count == (note_now * 2048) + 2047) rom_count <=
note_now * 2048;
                        else rom_count <= rom_count + 1;
```

```
                        write_count <= pointer;
                        pointer <= pointer;
                        state <= next;
                end
                else if (width_count == 31) begin
                        width_count <= 0;
                        block_count <= block_count;
                        length_count <= length_count + 1;
                        if (rom_count == (20 * 2048) + 2047) rom_count <= note_now *
2048;
                        else if (rom_count == (note_now * 2048) + 2047) rom_count <=
note_now * 2048;
                        else rom_count <= rom_count + 1;
                        write_count <= write_count + 1024 - 31;
                        pointer <= pointer;
                        state <= next;
                end
                else begin
                        block_count <= block_count;
                        width_count <= width_count + 1;
                        write_count <= write_count + 1;
                        if (rom_count == (20 * 2048) + 2047) rom_count <= note_now *
2048;
                        else if (rom_count == (note_now * 2048) + 2047) rom_count <=
note_now * 2048;
                        else rom_count <= rom_count + 1;
                        length_count <= length_count;
                        pointer <= pointer;
                        state <= next;
                end
        end
        else if (state == 6) begin
                width_count <= 0;
                block_count <= block_count;
                if (write_count == `V_ACTIVE * `H_ACTIVE) read_count <= 0;
                read_count <= read_count + 1;
                rom_count <= 0;
                length_count <= 0;
                pointer <= pointer;
                state <= next;
        end
        else if (state == 7) begin
                block_count <= block_count;
                write_count <= write_count;
                read_count <= read_count;
                if (block_count == 0) rom_count <= 20 * 2048;
```

```
                else rom_count <= note_now * 2048;
                width_count <= 0;
                length_count <= 0;
                pointer <= pointer;
                state <= next;
    end
        else begin
                block_count <= block_count;
                write_count <= write_count;
                read_count <= 0;
                rom_count <= 0;
                width_count <= 0;
                length_count <= 0;
                pointer <= pointer;
                state <= next;
        end
        end

    always @ *
        begin
        case (state)
                IDLE: begin
                        done_int <= 0;
                        we_int <= 0;
                        next <= SET;
                        ram_add <= write_count;
                        rom_add <= rom_count;
                        end
                SET:  begin
                                        done_int <= 0;
                        ram_add <= write_count;
                        rom_add <= rom_count;
                        if ((line_count == `V_ACTIVE - 1) && (pixel_count ==
`H_TOTAL - 3)) begin
                                next <= SETRAM;
                                we_int <= 0;
                        end
                        else begin
                                next <= SET;
                           we_int<= 0;
                        end
                        end
                SETRAM:      begin
                                        done_int <= 0;
                                 we_int<= 1;
                                 ram_add <= write_count;
```

```
                        rom_add <= rom_count;           //Modify for multiple
blocks
                        next <= SETROM;
                    end
            SETROM:  begin
                            done_int <= 0;
                        we_int<= 1;
                        ram_add <= write_count;
                    rom_add <= rom_count;           //Modify for multiple blocks
                        next <= WRITE;
                    end
            WRITE:   begin
                            done_int <= 0;
                    ram_add <= write_count;
                    rom_add <= rom_count;
                    if (ram_add == `H_ACTIVE * `V_ACTIVE - 1) begin
                            next <= WAIT;
                            we_int <= 0;
                    end
                    else begin
                            next <= WRITE;
                            we_int <= 1;
                    end
                    end
            WAIT:  begin
                            done_int <= 0;
                     we_int<= 0;
                    ram_add <= write_count;
                    rom_add <= rom_count;
                     if ((line_count == `V_TOTAL - 1) && (pixel_count ==
`H_TOTAL - 2)) next <= READ;
                     else next <= WAIT;
                     end
            READ:  begin
                            done_int <= 0;
                     we_int<= 0;
                    ram_add <= read_count;
                    rom_add <= rom_count;
                     if (pixel_count == `H_ACTIVE - 2) next <= READWAIT;
                     else next <= READ;
                     end
        READWAIT:    begin
                            done_int <= 0;
                   we_int <= 0;
                     ram_add <= read_count;
                     rom_add <= rom_count;
```

```
                        if ((line_count == `V_ACTIVE - 1) && (pixel_count ==
`H_TOTAL - 3)) begin
                                if (start && block_count == 0) next <=
TREBLERAM;
                                else if (start) next <= WRITERAM;
                                else next <= WAIT;
                        end
                        else if (pixel_count == `H_TOTAL - 2) next <= READ;
                        else next <= READWAIT;
                        end
        TREBLERAM:      begin
                         we_int<= 1;
                         ram_add <= write_count;
                         rom_add <= rom_count;          //Modify for multiple
blocks
                         done_int <= 0;
                         next <= TREBLEROM;
                        end
        TREBLEROM:      begin
                          we_int<= 1;
                          ram_add <= write_count;
                    rom_add <= rom_count;           //Modify for multiple blocks
                          next <= TREBLEWRITE;
                                done_int <= 0;
                        end
        TREBLEWRITE:    begin
                        ram_add <= write_count;
                        rom_add <= rom_count;
                        if ((length_count == 63) && (width_count == 31)) begin
                                next <= WRITING;
                                we_int <= 0;
                                done_int <= 1;
                        end
                        else begin
                                next <= TREBLEWRITE;
                                we_int <= 1;
                                done_int <= 0;
                        end
                        end

        WRITERAM: begin
                         we_int<= 1;
                         ram_add <= write_count;
                         rom_add <= rom_count;          //Modify for multiple
blocks
                         done_int <= 0;
```

```
                    next <= WRITEROM;
                end
        WRITEROM: begin
                    we_int<= 1;
                    ram_add <= write_count;
                rom_add <= rom_count;           //Modify for multiple blocks
                    next <= WRITING;
                        done_int <= 0;
                end
        WRITING:  begin
                ram_add <= write_count;
                rom_add <= rom_count;
                if ((length_count == 63) && (width_count == 31)) begin
                        next <= WAIT;
                        we_int <= 0;
                        done_int <= 1;
                end
                else begin
                        next <= WRITING;
                        we_int <= 1;
                        done_int <= 0;
                end
                end
        endcase
end
always @ (posedge pixel_clock)
begin

    if (color_sel == 0) begin
            vga_out_red <= (ram_out) ? 8'b00000000 : 8'b11111111;
            vga_out_green <= (ram_out) ? 8'b00000000 : 8'b11111111;
            vga_out_blue <= (ram_out) ? 8'b00000000 : 8'b11111111;
end
    else if (color_sel == 1) begin
            vga_out_red <= (ram_out) ? 8'b11111111 : 8'b00000000;
            vga_out_green <= (ram_out) ? 8'b11111111 : 8'b00000000;
            vga_out_blue <= (ram_out) ? 8'b11111111 : 8'b00000000;
end
    else if (color_sel == 2) begin
            vga_out_red <= (ram_out) ? 8'b00000000 : 8'b00000000;
            vga_out_green <= (ram_out) ? 8'b00000000 : 8'b00000000;
            vga_out_blue <= (ram_out) ? 8'b11111111 : 8'b00000000;
end
    else if (color_sel == 3) begin
            vga_out_red <= (ram_out) ? 8'b00000000 : 8'b00000000;
            vga_out_green <= (ram_out) ? 8'b11111111 : 8'b00000000;
```

```
                vga_out_blue <= (ram_out) ? 8'b00000000 : 8'b00000000;
    end
        end
endmodule
```