```
//adopted for Wake Up Your Way Alarm Clock
//by Eleanor Foltz and Jon Spurlock
//
//////////////////////////////////////////////////////////////////////////
////
//
// 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-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-Apr-29: Change history started
//
// 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-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-Oct-31: Adapted to new revision 004 board.
//
////////////////////////////////////////////////////////////////////////
////

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;
   // ac97_sdata_out and ac97_sdata_out are inputs;

   // 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;
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;
   // disp_data_out is an input   //no its not!!!

   // Buttons, Switches, and Individual LEDs
//   assign led = 8'hff;                    //LEDS off.
   // 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;

//synthesize attribute period of "clock_27mhz" is 37ns;
```

```
//code from here to end is not copied from website

wire hrflip, minflip, hzclock;
wire [17:0] clocktime,alarmtime;
wire imupbutton;
reg setalarm,setclock,lightSW,musicSW,voiceSW,buzzerSW,speedSW,alarmSW;
wire [7:0] sswitch;
wire [3:0] D_pad;
wire global_reset, msg_reset;
wire setfeature;
wire [1:0] featurenumber;
wire [5:0] earlylighttime,earlybuzzertime,earlymusictime,earlyvoicetime;
wire [5:0] earlytime;
reg [7:0] led;
wire [639:0] dots;
wire start_buzzer, stop_buzzer, buzzer_active,
                  start_light, stop_light, light_active,
                  start_music, stop_music, music_active,
                  start_voice, stop_voice, voice_active;
wire [7:0] buzzersound;
wire [5:0] max_light_int, light_int;
wire power;
wire st013_Reset, st013_SDA, st013_SCL;
wire [2:0] st013_status;

//createD_pad will synchronize these signals
createD_pad myD_pad(clock_27mhz, button_up, button_down, button_right,
button_left, D_pad);
synchronizer mysyncalarm(clock_27mhz, ~button_enter, imupbutton);
synchronizer mysyncrecord(clock_27mhz, ~button2, startrecord);
synchronizer mysyncmsgreset(clock_27mhz, ~button1, msg_reset);
synchronizer mysyncglobalreset(clock_27mhz, ~button0, global_reset);
synchronizer8bit mysync8bitswitch(clock_27mhz, switch, sswitch);

always @(posedge clock_27mhz) begin
      setalarm <= sswitch [7];
      setclock <= sswitch [6];
      lightSW  <= sswitch[5];
      musicSW  <= sswitch[4];
      voiceSW  <= sswitch[3];
      buzzerSW <= sswitch[2];
      speedSW <= sswitch[1];
      alarmSW  <= sswitch[0]; end

display my_display(global_reset, clock_27mhz,
```

```
                            disp_blank, disp_clock, disp_rs, disp_ce_b,
                            disp_reset_b, disp_data_out, dots);
displaymsg3 my_msg3(msg_reset, clock_27mhz, setfeature, setalarm,
featurenumber,
               earlytime, alarmtime, clocktime, dots);

clock_module2 myclock2(global_reset,clock_27mhz,setclock,D_pad,speedSW,
                     hrflip, minflip, hzclock, clocktime);
menu2 mymenu2(global_reset, clock_27mhz, D_pad,
            lightSW,musicSW,voiceSW,buzzerSW,

earlylighttime,earlybuzzertime,earlymusictime,earlyvoicetime,
                           featurenumber, setfeature, earlytime);
buzzermod mybuzzer(global_reset, clock_27mhz, start_buzzer, stop_buzzer,
                 buzzer_active, buzzersound);
light_mod mylight(clock_27mhz, global_reset, start_light, stop_light,
         max_light_int, light_int, light_active, power);

wire [2:0] major_state;
wire SDA_oen;
wire [7:0] rw_read_data;
wire st013_DATAREQ;
wire [10:0] rom_address;
music_major mymusic(clock_27mhz, global_reset, start_music, stop_music,
music_active,
         st013_Reset, SDA_oen, st013_SDA, st013_SCL, rw_read_data,
st013_status, st013_DATAREQ,
         major_state, rom_address);

//assign buzzer_active = 1'b0;
//assign light_active = 1'b0;
//assign music_active = 1'b0;
//assign voice_active = 1'b0;

alarm_top myalarm_top(clock_27mhz, global_reset, alarmSW,
                  alarmtime, setalarm, D_pad,
                  clocktime,    1'b0, //.clock_changed
                  earlybuzzertime,
                  earlylighttime,
                  earlymusictime,
                  earlyvoicetime,
                  start_buzzer, stop_buzzer, buzzer_active,
                  start_light, stop_light, light_active,
                  start_music, stop_music, music_active,
                  start_voice, stop_voice, voice_active,
                  max_light_int,
```

```
                  light_int);
//userX is 32bits
assign user4 = {23'bZZZZ_ZZZZ_ZZZZ_ZZZZ_ZZZZ_ZZZ,
         power,
         start_buzzer, stop_buzzer,
                   start_light, stop_light,
                   start_music, stop_music,
                   start_voice, stop_voice};

assign analyzer1_data = {8'hz, rw_read_data};
//assign analyzer3_data = {5'hz, st013_DATAREQ, rom_address};

wire st013_SDI, st013_SCKR;
assign st013_SDI = 0; assign st013_SCKR = 0;
assign user3 = {18'bZZZZ_ZZZZ_ZZZZ_ZZZZ_ZZ,
            start_music, SDA_oen,
            major_state,
         st013_status,
         st013_SDI, st013_SCKR, st013_DATAREQ,
         st013_SDA, st013_SCL,
         st013_Reset};

wire [7:0] audioout,voicesound,micsound;
wire CSbarDA,CSbarAD,RorWbar,status;
audio myaudio (global_reset, clock_27mhz, alarmSW,
               buzzersound, voicesound, //voicesound,
               audioout, CSbarDA);
assign user2 = {15'hZ,
               CSbarDA, audioout,
               8'hZ};
assign micsound = user1[31:24];
wire [3:0] voicestate;
wire [16:0] sramaddr;
voicemod myvoicemod(global_reset, clock_27mhz,
                start_voice, stop_voice, startrecord,
//                  1'b0,1'b0,startrecord,
                1'b1, micsound, CSbarAD, RorWbar, //status = 1'b1
                voicesound, voice_active, voicestate,sramaddr);
assign user1 = {8'hZ,
               status, CSbarAD, RorWbar,
               21'hZ};
assign analyzer3_data = {sramaddr[15:8],voicesound};
always @(posedge hzclock) begin
    if (global_reset) led <= ~(8'b0);     //all off
    else led <= {dots[639:638], sramaddr[14:13],micsound[2:0]};
    //{lightSW,musicSW,voiceSW,1'b0, ~clocktime[3:0]};
```

```
end

endmodule
```

```
//this code is adopted for final project
//
//author: Eleanor Foltz
//Synchronizer
//takes varin, which may change at any time,
//and returns varin_sync which is varin delayed
//to change value at 2nd posedge clk later.


module synchronizer (clk, varin, varin_sync);
input clk, varin;
output varin_sync;
reg varin_sync, temp_sync;

always @ (posedge clk) begin

    varin_sync <= temp_sync;
    temp_sync <= varin;

end //always posedge clk

endmodule
```

```
//Eleanor Foltz; Jon Spurlock;
// VERSION 1.0

module createD_pad (clock_27mhz, button_up, button_down, button_right,
button_left, D_pad);

input clock_27mhz, button_up, button_down, button_right, button_left;
output [3:0] D_pad;

wire [3:0]currentD;
reg [3:0] lastD, D_pad;

wire button_up_sync, button_down_sync, button_right_sync, button_left_sync;

synchronizer mysyncup(clock_27mhz, button_up, button_up_sync);
synchronizer mysyncdown(clock_27mhz, button_down, button_down_sync);
synchronizer mysyncright(clock_27mhz, button_right, button_right_sync);
synchronizer mysyncleft(clock_27mhz, button_left, button_left_sync);

assign currentD =
{button_up_sync,button_down_sync,button_right_sync,button_left_sync};

always @(posedge clock_27mhz) begin
   lastD <= currentD;
   D_pad <= (4'b0 ^ currentD) & (lastD ^ currentD);  //bitwise XOR,
bitwise &
end


endmodule
```

```
// VERSION 1.0 approx 5/7/05
//Version 1.1
//incorporates speedSW for divider 5/9 4pm

module clock_module2(reset, clock_27mhz, setclock,D_pad, speedSW,
                     hrflip, minflip, hzclock, clocktime);

input reset, clock_27mhz, setclock, speedSW;
input [3:0] D_pad;
output hrflip, minflip, hzclock;
output [17:0] clocktime;

reg [17:0] clocktime;
wire [17:0] time_next, time_from_changefsm;
reg last_hzclock;
reg changing;
wire hours_carry;
parameter add = 2'b00; //parameter subtract = 2'b01;

divider mydivider(reset, clock_27mhz, hzclock, speedSW);

time_alu my_clockt_alu(add, clocktime, 18'b1, time_next, hours_carry);

clocktimefsm myclockchanger(reset, clock_27mhz, changing, clocktime,
                        D_pad, time_from_changefsm);

always @ (posedge clock_27mhz) begin
    last_hzclock <= hzclock;
    if (reset) clocktime <= 18'b0;
    else begin
        if (setclock)  begin
            changing = 1'b1;
            clocktime <= time_from_changefsm;
        end
        else begin
            changing = 1'b0;
            if (last_hzclock == hzclock);
            else begin if (hzclock) clocktime <= time_next; end end
    end
end

assign minflip = ~(clocktime[6] == time_next[6]);  // compares lsb of
minutes
assign hrflip = ~(clocktime[12] == time_next[12]);  // compares lsb of
hours
```

```
endmodule
```

```
//Eleanor Foltz; Jon Spurlock;
// VERSION 1.0

module clocktimefsm(reset, clk, changing, current_time,
                     D_pad, newtime);

input clk, reset, changing;
input [3:0] D_pad;
input [17:0] current_time;
output [17:0] newtime;
//output [2:0] state; //for debug
reg  [17:0] newtime;
wire [17:0] earliertime, latertime;

reg [2:0] state;

parameter RESET=7;parameter IDLE=0; parameter SET_TIME=1;
parameter EARLIER=2; parameter LATER = 3;
parameter EARLIER2=4; parameter LATER2 = 5;
parameter up_button_bit = 3;parameter down_button_bit = 2;
parameter        add = 2'b0; parameter   subtract= 2'b1;

wire hours_carry1,hours_carry2;

time_alu myearliertime(subtract, current_time, 18'b000000_000001_000000,
                       earliertime, hours_carry1);
time_alu mylatertime(add, current_time, 18'b000000_000001_000000,
                       latertime, hours_carry2);

always @(posedge clk) begin
    if (reset) begin
        state <= RESET;
    end
    else begin
        case (state)
           default: ;
           RESET: begin
               newtime <= 18'b0;
               state <= IDLE;
           end
           IDLE: begin
               state <= changing ? SET_TIME : IDLE;
               newtime <= current_time;
           end
           SET_TIME: begin
               if (changing == 0)
```

```
                state <= IDLE;
            else if (D_pad[up_button_bit]==1)
                state <= LATER;
            else if (D_pad[down_button_bit]==1)
                state <= EARLIER;
            else state <= SET_TIME;
        end
        EARLIER: begin
            newtime <= earliertime;
            state <= EARLIER2;
        end
        EARLIER2: begin
            state <= (D_pad[down_button_bit]==1) ? EARLIER2 : SET_TIME;
        end
        LATER: begin
            newtime <= latertime;
            state <= LATER2;
        end
        LATER2: begin
            state <= (D_pad[up_button_bit]==1) ? LATER2 : SET_TIME;
        end
    endcase
  end //else
end //always


endmodule
```

```
// Jon Spurlock; Eleanor Foltz; 6.111 sp05 final project
//written by JS, 4/13/05
//EF added subtract mode on 4/15

// VERSION 1.0

module time_alu (mode, time_in1, time_in2, time_out, hours_carry);

   //times are 18 bits - 6 lsb's for seconds in bin, 6 middle bits for
   //minutes in bin, next 5 bits for hours in bin, and msb could be used
   //for am vs pm, unused at this time

   input [1:0] mode;     //needed? could also be used for 12 vs 24 hour
mode
   input [17:0] time_in1, time_in2;

   output       hours_carry;
   output [17:0] time_out;
   //output      time_out_seconds; //need to figure out bit # in 24*60*60

   reg [17:0]    time_out;
   //reg                 time_out_seconds; //bit #?

   reg [6:0]     seconds_subtot;
   reg [6:0]     minutes_subtot;
   reg [5:0]     hours_subtot;

   reg           seconds_carry, minutes_carry, hours_carry;

   // 24 hour time
   parameter     add = 2'b0;
   parameter     subtract= 2'b1;



   always @ (mode or time_in1 or time_in2)

     case (mode)
       default: ;
       add:     //add time_in1 and time_in2
         begin

            seconds_subtot = time_in1[5:0] + time_in2[5:0];
            if (seconds_subtot > 59)
              begin
                        seconds_subtot = seconds_subtot - 60;
```

```
                        time_out[5:0] = seconds_subtot[5:0];
                        seconds_carry = 1;
              end
            else
              begin
                        time_out[5:0] = seconds_subtot[5:0];
                        seconds_carry = 0;
              end // else: !if(seconds_subtot > 59)

            minutes_subtot = time_in1[11:6] + time_in2[11:6] +
seconds_carry;
            if (minutes_subtot > 59)
              begin
                        minutes_subtot = minutes_subtot - 60;
                        time_out[11:6] = minutes_subtot[5:0];
                        minutes_carry = 1;
              end
            else
              begin
                        time_out[11:6] = minutes_subtot[5:0];
                        minutes_carry = 0;
              end // else: !if(minutes_subtot > 59)

            hours_subtot = time_in1[16:12] + time_in2[16:12] +
minutes_carry;
            if (hours_subtot > 23)
              begin
                        hours_subtot = hours_subtot - 24;
                        time_out[16:12] = hours_subtot[4:0];
                        hours_carry = 1;
              end
            else
              begin
                        time_out[16:12] = hours_subtot[4:0];
                        hours_carry = 0;
              end // else: !if(hours-subtot > 23)

            time_out[17] = 0; //could be used for am vs pm in 12 hour mode,
                              //assuming here remains 0 in 24 hour mode

         end // case: add

        subtract:     //subtract time_in1 and time_in2
          begin

            seconds_subtot = time_in1[5:0] - time_in2[5:0];
```

```
            if (seconds_subtot[6] ==1)
              begin
                        seconds_subtot = seconds_subtot + 60;
                        time_out[5:0] = seconds_subtot[5:0];
                        seconds_carry = 1;
              end
            else
              begin
                        time_out[5:0] = seconds_subtot[5:0];
                        seconds_carry = 0;
              end // else: !if(seconds_subtot <0)

            minutes_subtot = time_in1[11:6] - time_in2[11:6] -
seconds_carry;
            if (minutes_subtot[6] ==1)
              begin
                        minutes_subtot = minutes_subtot + 60;
                        time_out[11:6] = minutes_subtot[5:0];
                        minutes_carry = 1;
              end
            else
              begin
                        time_out[11:6] = minutes_subtot[5:0];
                        minutes_carry = 0;
              end // else: !if(minutes_subtot <0)

            hours_subtot = time_in1[16:12] - time_in2[16:12] -
minutes_carry;
            if (hours_subtot[5]==1)
              begin
                        hours_subtot = hours_subtot + 24;
                        time_out[16:12] = hours_subtot[4:0];
                        hours_carry = 1;
              end
            else
              begin
                        time_out[16:12] = hours_subtot[4:0];
                        hours_carry = 0;
              end // else: !if(hours-subtot <0)

            time_out[17] = 0; //could be used for am vs pm in 12 hour mode,
                              //assuming here remains 0 in 24 hour mode

          end // case: add
        endcase // case(mode)
```

```
//assign time_out_seconds here ...

endmodule // time_alu
```

```
//Eleanor Foltz; Jon Spurlock;
//1hz clock output

// VERSION 1.0
//Version 1.1
// added speed switch

module divider(reset, clock_27mhz, hzclock, speedSW);
input reset, clock_27mhz, speedSW;

output hzclock;
reg hzclock;
reg [25:0] partseconds_count;  //unsigned magnitude only

always @ (posedge clock_27mhz) begin
    if (reset) begin
        partseconds_count <= 0;
        hzclock <=0;
    end
    else begin
        if (speedSW) begin
           if (partseconds_count == 800000) begin
               partseconds_count <= 0;
               hzclock <= ~hzclock; end
           else partseconds_count <= partseconds_count + 1; end
        else if (partseconds_count == 13499999) begin
               partseconds_count <= 0;
               hzclock <= ~hzclock; end
           else partseconds_count <= partseconds_count + 1;
    end
end //always


endmodule
```

```
// Jon Spurlock; Eleanor Foltz; 6.111 sp05 final project

// VERSION 1.1
// added my line 63

module alarm_top (clk,
                  rst,

                  alarm_status,

                  alarm_time,
                  set,
                  D_pad,

                  clock_time,
                  clock_changed,

                  buzzer_dt,
                  light_dt,
                  music_dt,
                  voice_dt,

                  start_buzzer, stop_buzzer, buzzer_active,
                  start_light, stop_light, light_active,
                  start_music, stop_music, music_active,
                  start_voice, stop_voice, voice_active,

                  max_light_int,
                  light_int);

   input clk, rst;

   input alarm_status;

   input set;
   input [3:0] D_pad;

   output [17:0] alarm_time;
   wire [17:0]   alarm_time;

   wire          alarm_changed;

   input [17:0] clock_time;
   input        clock_changed;

   input [5:0]  buzzer_dt, light_dt, music_dt, voice_dt;
```

```
wire         submod_start, submod_stop;
wire [17:0]  buzzer_time, light_time, music_time, voice_time;
wire         buzzer_done, light_done, music_done, voice_done;

output       start_buzzer, stop_buzzer;
output       start_light, stop_light;
output       start_music, stop_music;
output       start_voice, stop_voice;

input        buzzer_active, light_active, music_active, voice_active;

output [5:0] max_light_int;
output [5:0] light_int;


alarm_time myalarm_time (.clk (clk),
                         .rst (rst),
                         .set (set),
                         .dpad (D_pad),
                         .alarm (alarm_time),
                         .changed (alarm_changed));

alarm_main main (.clk (clk),
                 .rst (rst),
                 .alarm_time (alarm_time),
                 .alarm_changed (alarm_changed),
                 .clock_time (clock_time),
                 .clock_changed (clock_changed),
                 .buzzer_dt (buzzer_dt),
                 .light_dt (light_dt),
                 .music_dt (music_dt),
                 .voice_dt (voice_dt),
                 .alarm_status (alarm_status),
                 .buzzer_done (buzzer_done),
                 .light_done (light_done),
                 .music_done (music_done),
                 .voice_done (voice_done),
                 .buzzer_time (buzzer_time),
                 .light_time (light_time),
                 .music_time (music_time),
                 .voice_time (voice_time),
                 .submod_start (submod_start),
                 .submod_stop (submod_stop));

gen_alarm_mod buzzer_mod (.clk (clk),
```

```
                            .rst (rst),
                            .gen_mod_time (buzzer_time),
                            .clock_time (clock_time),
                            .start_mod (submod_start),
                            .stop_mod (submod_stop),
                            .done (buzzer_done),
                            .start_function (start_buzzer),
                            .stop_function (stop_buzzer),
                            .function_active (buzzer_active));

    light_alarm_mod light_mod (.clk (clk),
                            .rst (rst),
                            .mod_time (light_time),
                            .clock_time (clock_time),
                             .alarm_time (alarm_time),
                            .start_mod (submod_start),
                            .stop_mod (submod_stop),
                            .done (light_done),
                            .start_function (start_light),
                            .stop_function (stop_light),
                             .max_intensity (max_light_int),
                             .intensity (light_int),
                            .function_active (light_active));

    gen_alarm_mod music_mod (.clk (clk),
                            .rst (rst),
                            .gen_mod_time (music_time),
                            .clock_time (clock_time),
                            .start_mod (submod_start),
                            .stop_mod (submod_stop),
                            .done (music_done),
                            .start_function (start_music),
                            .stop_function (stop_music),
                            .function_active (music_active));

    gen_alarm_mod voice_mod (.clk (clk),
                            .rst (rst),
                            .gen_mod_time (voice_time),
                            .clock_time (clock_time),
                            .start_mod (submod_start),
                            .stop_mod (submod_stop),
                            .done (voice_done),
                            .start_function (start_voice),
                            .stop_function (stop_voice),
                            .function_active (voice_active));
```

```
endmodule // alarm_top
```

```
// Jon Spurlock; Eleanor Foltz; 6.111 sp05 final project

// VERSION 1.0

module alarm_main (clk,
                   rst,

                   alarm_time,
                   alarm_changed,

                   clock_time,
                   clock_changed,

                   buzzer_dt,
                   light_dt,
                   music_dt,
                   voice_dt,

                   alarm_status,

                   buzzer_done,
                   light_done,
                   music_done,
                   voice_done,

                   buzzer_time,
                   light_time,
                   music_time,
                   voice_time,

                   submod_start,
                   submod_stop);

   input clk, rst;

   input [17:0] alarm_time, clock_time;
   input        alarm_changed, clock_changed;

   input [5:0]  buzzer_dt, light_dt, music_dt, voice_dt;

   input        alarm_status;

   input        buzzer_done, light_done, music_done, voice_done;
   wire         modules_done;

   output       buzzer_time, light_time, music_time, voice_time;
```

```
output       submod_start, submod_stop;

wire [17:0]  buzzer_time, light_time, music_time, voice_time;
reg          submod_start, next_substart, submod_stop, next_substop;
reg [2:0]    state, next_state;

wire [17:0]  time_delta;
reg          alarm_trig;

wire         hrs_carry1, hrs_carry2, hrs_carry3, hrs_carry4, hrs_carry5;

parameter    subtract = 2'b01;

parameter    off_state = 0;
parameter    wait_state = 1;
parameter    alarm_state = 2;
parameter    reset_state = 3;


time_alu time_diff (.mode (subtract),
                    .time_in1 (alarm_time),
                    .time_in2 (clock_time),
                    .time_out (time_delta),
                    .hours_carry (hrs_carry1));


/* may need to register inputs for the sake of value sampling:

always @ (posedge clk)
   begin
       */


always @ (time_delta)
  begin

     if (time_delta[16:12] == 0 && time_delta[11:6] <= 30)
       begin
          alarm_trig = 1;
       end
     else
       begin
          alarm_trig = 0;
       end
```

```
  end // always @ (alarm_time or clock_time)


 time_alu buzzer_conv (.mode ({1'b0, buzzer_dt[5]}),
                       .time_in1 (alarm_time),
                       .time_in2 ({7'b0000000, buzzer_dt[4:0], 6'b000000}),
                       .time_out (buzzer_time),
                       .hours_carry (hrs_carry2));

 time_alu light_conv (.mode ({1'b0, light_dt[5]}),
                       .time_in1 (alarm_time),
                       .time_in2 ({7'b0000000, light_dt[4:0], 6'b000000}),
                       .time_out (light_time),
                       .hours_carry (hrs_carry3));


 time_alu music_conv (.mode ({1'b0, music_dt[5]}),
                       .time_in1 (alarm_time),
                       .time_in2 ({7'b0000000, music_dt[4:0], 6'b000000}),
                       .time_out (music_time),
                       .hours_carry (hrs_carry4));

 time_alu voice_conv (.mode ({1'b0, voice_dt[5]}),
                       .time_in1 (alarm_time),
                       .time_in2 ({7'b0000000, voice_dt[4:0], 6'b000000}),
                       .time_out (voice_time),
                       .hours_carry (hrs_carry5));

/* always @ (alarm_time or buzzer_dt or light_dt or music_dt or voice_dt)
   begin

      buzzer_time = alarm_time + buzzer_dt;
      light_time = alarm_time + light_dt;
      music_time = alarm_time + music_dt;
      voice_time = alarm_time + voice_dt;

   //end */


 always @ (posedge clk)
   begin

      if (rst || !alarm_status)
        begin
           state <= off_state;
           submod_start <= 0;
```

```
            submod_stop <= 1;
          end
        else if (clock_changed || alarm_changed)
          begin
             state <= state;
             submod_start <= 0;
             submod_stop <= submod_stop;
          end
        else
          begin
             state <= next_state;
             submod_start <= next_substart;
             submod_stop <= next_substop;
          end // else: !if(!rst or !alarm_status)



     end // always @ (posedge clk)


   assign modules_done = buzzer_done && light_done && music_done &&
voice_done;

   always @ (state or alarm_trig or modules_done or time_delta)
     begin

        case (state)

          off_state:
            begin
               next_state = wait_state;
               next_substart = 0;
               next_substop = 1;
            end

          wait_state:
            begin

               if (alarm_trig)
                 begin
                    next_state = alarm_state;
                    next_substart = 0;
                    next_substop = 0;
                 end
               else
                 begin
```

```
                next_state = wait_state;
                next_substart = 0;
                next_substop = 1;
              end // else: !if(alarm_trig)

        end // case: wait_state

      alarm_state:
        begin

           if (modules_done)
             begin
                next_state = reset_state;
                next_substart = 0;
                next_substop = 0;
             end
           else
             begin
                next_state = alarm_state;
                next_substart = 1;
                next_substop = 0;
             end // else: !if(modules_done)

        end // case: alarm_state

      reset_state:
        begin

           next_substart = 0;
           next_substop = 1;

           if (time_delta[16:12] >= 1)
             begin
                next_state = wait_state;
             end
           else
             begin
                next_state = reset_state;
             end

        end // case: reset_state

    endcase // case(state)

end // always @ (state)
```

```
endmodule // alarm_top
```

```
// Jon Spurlock; Eleanor Foltz; 6.111 sp05 final project

// VERSION 1.0
// VERSION 1.1 added one am default

module alarm_time (clk, rst, set, dpad, alarm, changed);

   input clk, rst, set;
   input [3:0] dpad;

   output      changed;
   output [17:0] alarm;

   reg           changed;
   reg [17:0]    alarm;

   wire [17:0] alarm_inc, alarm_dec;

   wire hours_carry; // not used
   wire hours_carry2;

   parameter   oneam = 18'b000001000000000000; //for test
   parameter     midnight = 18'b000000000000000000;
   parameter   threeam = 18'b000011000000000000;  //for test
   parameter   sixam = 18'b000110000000000000;  //for test
   parameter   noon = 18'b001100000000000000;  //for test
   parameter     minute = 18'b000000000001000000;

   parameter     dpad_up = 4'b1000;
   parameter     dpad_down = 4'b0100;
   parameter     dpad_left = 4'b0010;
   parameter     dpad_right = 4'b0001;

   parameter add = 2'b00;
   parameter subtract = 2'b01;


   always @ (posedge clk)
     begin

        if (rst)
          begin
             alarm <= oneam;
             changed <= 1;
          end
        else if (set)
```

```
          begin
             if (dpad == dpad_up)
               begin
                  alarm <= alarm_inc;
                  changed <= 1;
               end
             else if (dpad == dpad_down)
               begin
                  alarm <= alarm_dec;
                  changed <= 1;
               end
             else
               begin
                  alarm <= alarm;
                  changed <= changed;
               end
          end // if (set)
        else
          begin
             alarm <= alarm;
             changed <= 0;
          end // else: !if(set)

      end // always @ (posedge clk)

   time_alu time_inc (add, alarm, minute, alarm_inc, hours_carry);
   time_alu time_dec (subtract, alarm, minute, alarm_dec, hours_carry2);

endmodule // alarm_time
```

```
// Jon Spurlock; Eleanor Foltz; 6.111 sp05 final project

// VERSION 1.0
// VERSION 1.1 modified for persistent 'start_function'

module gen_alarm_mod (clk,
                      rst,

                      gen_mod_time,
                      clock_time,

                      start_mod,
                      stop_mod,

                      done,

                      start_function,
                      stop_function,

                      function_active);

   input clk, rst;

   input [17:0] gen_mod_time, clock_time;

   input        start_mod, stop_mod;

   output       done;

   output       start_function, stop_function;

   input        function_active;

   reg          done, next_done;
   reg          start_function, next_funstart, stop_function, next_funstop;

   reg [2:0]    state, next_state;


   parameter    idle_state = 0;
   parameter    start_state = 1;
   parameter    trig_state = 2;
   parameter    alarm_state = 3;
   //parameter  stop_state = 4;
```

```
   always @ (posedge clk)
     begin

        if (rst || stop_mod)
          begin
             state <= idle_state;
             start_function <= 0;
             stop_function <= 1;
             done <= 0;
          end

        else
          begin
             state <= next_state;
             start_function <= next_funstart;
             stop_function <= next_funstop;
             done <= next_done;
          end // else: !if(!rst or stop_mod)

     end // always @ (posedge clk)


   always @ (state or start_mod or stop_mod or function_active or
clock_time or gen_mod_time)
     begin

        case (state)

          idle_state:
            begin

               if (start_mod)
                 begin
                    next_state = start_state;
                    next_funstart = 0;
                    next_funstop = 0;
                    next_done = 0;
                 end

               else
                 begin
                    next_state = idle_state;
                    next_funstart = 0;
                    next_funstop = 1;
                    next_done = 0;
                 end // else: !if(start_mod)
```

```
  end // case: idle_state

start_state:
  begin

     if (clock_time >= gen_mod_time)
       begin
          next_state = trig_state;
          next_funstart = 1;
          next_funstop = 0;
          next_done = 0;
       end

     else
       begin
          next_state = start_state;
          next_funstart = 0;
          next_funstop = 0;
          next_done = 0;
       end // else: !if(clock_time >= gen_mod_time)

  end // case: start_state

trig_state:
  begin

     if (function_active)
       begin
          next_state = alarm_state;
          next_funstart = 1;
          next_funstop = 0;
          next_done = 0;
       end

     else
       begin
          next_state = trig_state;
          next_funstart = 1;
          next_funstop = 0;
          next_done = 0;
       end // else: !if(function_active)

  end // case: trig_state

alarm_state:
```

```
            begin

               if (!start_mod)
                 begin
                    next_state = idle_state;
                    next_funstart = 0;
                    next_funstop = 0;
                    next_done = 0;
                 end

               else if (!function_active)
                 begin
                    next_state = alarm_state;
                    next_funstart = 0;
                    next_funstop = 0;
                    next_done = 1;
                 end

               else
                 begin
                    next_state = alarm_state;
                    next_funstart = 1;
                    next_funstop = 0;
                    next_done = 0;
                 end // else: !if(!function_active)

            end // case: alarm_state

          endcase // case (state)

        end // always @ (state or start_mod or stop_mod or function_status)

endmodule // gen_alarm_mod
```

```
// Jon Spurlock; Eleanor Foltz; 6.111 sp05 final project

// VERSION 1.0
//version 1.1  uncommented hourscarry.

module light_alarm_mod (clk,
                        rst,

                        mod_time,
                        clock_time,
                          alarm_time,

                        start_mod,
                        stop_mod,

                        done,

                        start_function,
                        stop_function,

                          max_intensity,
                          intensity,

                        function_active);

   input clk, rst;

   input [17:0] mod_time, clock_time, alarm_time;

   input        start_mod, stop_mod;

   output       done;

   output       start_function, stop_function;

   output [5:0] max_intensity;
   output [5:0] intensity;

   input        function_active;

   reg          done, next_done;
   reg          start_function, next_funstart, stop_function, next_funstop;

   reg [5:0]    max_intensity, intensity;

   reg [2:0]    state, next_state;
```

```
wire [17:0]  alarm_plus5, max_intensity_all, intensity_all;
wire hrs_carry1, hrs_carry2, hrs_carry3;

parameter    idle_state = 0;
parameter    start_state = 1;
parameter    trig_state = 2;
parameter    alarm_state = 3;

parameter    subtract = 2'd1;
parameter    add = 2'd0;
parameter    five_minutes = {6'd0,6'd5,6'd0};


time_alu alarm_plus (.mode (add),
                     .time_in1 (alarm_time),
                     .time_in2 (five_minutes),
                     .time_out (alarm_plus5) ,
                     .hours_carry (hrs_carry1) );

time_alu max_int (.mode (subtract),
                  .time_in1 (alarm_plus5),
                  .time_in2 (mod_time),
                  .time_out (max_intensity_all) ,
                  .hours_carry (hrs_carry2) );

time_alu int (.mode (subtract),
              .time_in1 (clock_time),
              .time_in2 (mod_time),
              .time_out (intensity_all) ,
              .hours_carry (hrs_carry3) );


always @ (posedge clk)
  begin

     if (rst || stop_mod)
       begin
          state <= idle_state;
          start_function <= 0;
          stop_function <= 1;
          done <= 0;
       end

     else
       begin
```

```
            state <= next_state;
            start_function <= next_funstart;
            stop_function <= next_funstop;
            done <= next_done;
          end // else: !if(!rst or stop_mod)

        max_intensity = max_intensity_all[11:6];
        intensity = intensity_all[11:6];

     end // always @ (posedge clk)


   always @ (state or start_mod or stop_mod or function_active or
clock_time or mod_time)
     begin

        case (state)

          idle_state:
            begin

               if (start_mod)
                 begin
                    next_state = start_state;
                    next_funstart = 0;
                    next_funstop = 0;
                    next_done = 0;
                 end

               else
                 begin
                    next_state = idle_state;
                    next_funstart = 0;
                    next_funstop = 1;
                    next_done = 0;
                 end // else: !if(start_mod)

            end // case: idle_state

          start_state:
            begin

               if (clock_time >= mod_time)
                 begin
                    next_state = trig_state;
                    next_funstart = 1;
```

```
            next_funstop = 0;
            next_done = 0;
          end

        else
          begin
            next_state = start_state;
            next_funstart = 0;
            next_funstop = 0;
            next_done = 0;
          end // else: !if(clock_time >= mod_time)

    end // case: start_state

  trig_state:
    begin

        if (function_active)
          begin
            next_state = alarm_state;
            next_funstart = 1;
            next_funstop = 0;
            next_done = 0;
          end

        else
          begin
            next_state = trig_state;
            next_funstart = 1;
            next_funstop = 0;
            next_done = 0;
          end // else: !if(function_active)

    end // case: trig_state

  alarm_state:
    begin

        if (!start_mod)
          begin
            next_state = idle_state;
            next_funstart = 0;
            next_funstop = 0;
            next_done = 0;
          end
```

```
                else if (!function_active)
                  begin
                     next_state = alarm_state;
                     next_funstart = 0;
                     next_funstop = 0;
                     next_done = 1;
                  end

                else
                  begin
                     next_state = alarm_state;
                     next_funstart = 1;
                     next_funstop = 0;
                     next_done = 0;
                  end // else: !if(!function_active)

              end // case: alarm_state

            endcase // case (state)

          end // always @ (state or start_mod or stop_mod or function_status)

endmodule // light_alarm_mod
```

```
//Eleanor Foltz; Jon Spurlock;

//saving as menu2 because of significant modifications 4/29/05 EF

module menu2(global_reset, clk, D_pad,
            lightSW,musicSW,voiceSW,buzzerSW,

earlylighttime,earlybuzzertime,earlymusictime,earlyvoicetime,
                           featurenumber, setfeature, earlytime);

input global_reset, clk;
input [3:0] D_pad;
input lightSW,musicSW,voiceSW,buzzerSW;
output [5:0] earlylighttime,earlybuzzertime,earlymusictime,earlyvoicetime;
output [1:0] featurenumber;
output setfeature;
output [5:0] earlytime;
wire setlight, setbuzzer, setmusic, setvoice;
reg [5:0] earlytime;
parameter thirtymin = {6'b100010};  //6'b111110;
menuchoose mymenuchoose(clk, featurenumber,
                                            lightSW, buzzerSW, musicSW,
voiceSW,
                  setlight, setbuzzer, setmusic, setvoice);

smallmenufsm lightmenufsm (clk, global_reset, setlight, thirtymin,
6'b001010,
                      D_pad,     earlylighttime);
smallmenufsm buzzermenufsm (clk, global_reset, setbuzzer, thirtymin,
6'b011110,
                      D_pad,     earlybuzzertime);
smallmenufsm musicmenufsm (clk, global_reset, setmusic, thirtymin,
6'b001010,
                      D_pad,     earlymusictime);
smallmenufsm voicemenufsm (clk, global_reset, setvoice, thirtymin,
6'b001010,
                      D_pad,     earlyvoicetime);

assign setfeature = setlight || setbuzzer || setmusic || setvoice;

always @ (posedge clk) begin
    case (featurenumber)
        default: earlytime <= 6'b0;
        3: earlytime <= earlybuzzertime;
        0: earlytime <= earlylighttime;
        1: earlytime <= earlymusictime;
```

```
        2: earlytime <= earlyvoicetime;
   endcase
end

endmodule
```

```
//Eleanor Foltz; Jon Spurlock;

//this module is contained within the menu block
//its only purpose is to ensure that only one of
//the small menu FSMs are triggered at any time.

module menuchoose(clk, featurenumber,
                  setlightSW, setbuzzerSW, setmusicSW, setvoiceSW,
                  setlight, setbuzzer, setmusic, setvoice);

input clk, setlightSW, setbuzzerSW, setmusicSW, setvoiceSW;
output [1:0] featurenumber;
output setlight, setbuzzer, setmusic, setvoice;

reg setlight, setbuzzer, setmusic, setvoice;
reg [1:0] featurenumber;

always @(posedge clk) begin
    if (setbuzzerSW == 1) begin
        setbuzzer<=1; setlight<=0; setmusic<=0; setvoice<=0;
        featurenumber <= 2'b11; end
    else if (setlightSW == 1) begin
        setbuzzer<=0; setlight<=1; setmusic<=0; setvoice<=0;
        featurenumber <= 2'b00;  end
    else if (setmusicSW == 1) begin
        setbuzzer<=0; setlight<=0; setmusic<=1; setvoice<=0;
        featurenumber <= 2'b01; end
    else if (setvoiceSW == 1) begin
        setbuzzer<=0; setlight<=0; setmusic<=0; setvoice<=1;
        featurenumber <= 2'b10; end
    else begin
        setbuzzer<=0; setlight<=0; setmusic<=0; setvoice<=0;
        featurenumber <= 2'b01; end
end //always
endmodule
```

```
//Eleanor Foltz; Jon Spurlock;

module smallmenufsm(clk, reset, setfeature, tmin, tmax,
                     D_pad,     earlytime);

input clk, reset, setfeature;
input [3:0] D_pad;
input [5:0] tmin, tmax;
//tmin and tmax are 2s complement times in minutes
//so they are each 7 bits long;
output [5:0] earlytime;

reg [5:0] earlytimetwos;
reg [2:0] state;

parameter RESET=0; parameter IDLE=1; parameter SET_TIME=2;
parameter EARLIER=3; parameter LATER = 5;
parameter EARLIERWAIT=4; parameter LATERWAIT = 6;
parameter up_button_bit = 3;parameter down_button_bit = 2;

twosTOsignmag mytwo_signmag(earlytimetwos, earlytime);

always @(posedge clk) begin
    if (reset == 1) state <= RESET;
    else case (state)
       default: state <= RESET;
       RESET: begin
           earlytimetwos <= 6'b0;
           state <= IDLE;
       end
       IDLE:  state <= setfeature? SET_TIME : IDLE;
       SET_TIME: begin
           if (setfeature == 0)
              state <= IDLE;
           else if (D_pad[up_button_bit]==1)
              state <= LATER;
           else if (D_pad[down_button_bit]==1)
              state <= EARLIER;
           else state <= SET_TIME;
       end
       LATER: begin
           if (earlytimetwos[5] == tmax[5])
             begin if (earlytimetwos[4:0] < tmax[4:0])
                earlytimetwos <= earlytimetwos + 1; end
           else earlytimetwos <= earlytimetwos + 1;
           state <= LATERWAIT;
```

```
        end
        LATERWAIT: begin
            state <= (D_pad[up_button_bit]==1) ? LATERWAIT : SET_TIME;
        end
        EARLIER: begin
            if (earlytimetwos[5] == tmin[5])
              begin if (earlytimetwos[4:0] > tmin[4:0])
                 earlytimetwos <= earlytimetwos - 1; end
            else earlytimetwos <= earlytimetwos - 1;
            state <= EARLIERWAIT;
        end
        EARLIERWAIT: begin
            state <= (D_pad[down_button_bit]==1) ? EARLIERWAIT : SET_TIME;
        end
    endcase
end //always

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) begin
     if (reset)
       reset_count <= 100;
     else
       reset_count <= (reset_count==0) ? 0 : reset_count-1;
```

```
   end //always

   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) begin
     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
 end //always
endmodule
```

```
//changing name of module to "3" because of major modification
//  to include display alarm time.  4/29/05  EF
module displaymsg3(reset, clock, setfeature, setalarm, featurenumber,
               earlytime, alarmtime, clocktime, dots);

input reset, clock, setfeature, setalarm;
input [1:0] featurenumber;
input [5:0] earlytime;
input [17:0] clocktime, alarmtime;
output [639:0] dots;
 //earlytime in minutes, sign mag + 5bits for 31.

parameter displayA = 40'b01111000_00010110_00010011_00010110_01111000; //
'A'
parameter displayB = 40'b01111111_01001001_01001001_01001001_00110110; //
'B'
parameter displayC = 40'b00111110_01000001_01000001_01000001_00100010; //
'C'
parameter displayE = 40'b01111111_01001001_01001001_01001001_01001001; //
'E'
parameter displayG = 40'b00111110_01000001_01010001_00110001_01010010; //
'G'
parameter displayH = 40'b01111111_00001000_00001000_00001000_01111111; //
'H'
parameter displayI = 40'b01000001_01000001_01111111_01000001_01000001; //
'I'
parameter displayK = 40'b01111111_00001000_00010100_00100010_01000001; //
'K'
parameter displayL = 40'b01111111_01000000_01000000_01000000_01000000; //
'L'
parameter displayM = 40'b01111111_00000010_00001100_00000010_01111111; //
'M'
parameter displayN = 40'b01111111_00000110_00001100_00110000_01111111; //
'N'
parameter displayP = 40'b01111111_00001001_00001001_00001001_00000110; //
'P'
parameter displayR = 40'b01111111_00001001_00011001_01101001_01000110; //
'R'
parameter displayS = 40'b01000110_01001001_01001001_01001001_00110001; //
'S'
parameter displayT = 40'b00000001_00000001_01111111_00000001_00000001; //
'T'
parameter displayU = 40'b00111111_01000000_01000000_01000000_00111111; //
'U'
parameter displayV = 40'b00000111_00011000_01100000_00011000_00000111; //
'V'
```

```
parameter displayZ = 40'b01000100_01100100_01010100_01001100_01000100; //
'Z'
parameter displayspace = 40'b00000000_00000000_00000000_00000000_00000000;
// ' '
parameter displaycolon = 40'b00000000_00110110_00110110_00000000_00000000;
// ':'
parameter displaydash = 40'b00000000_00001100_00001100_00001100_00000000; /
/ '-'
parameter display0 = 40'b00111110_01000001_01000001_01000001_00111110; //
'0'

reg [39:0] char[15:0];

wire [39:0] fmin1, fmin0, csec1,csec0, cmin1,cmin0, chr1,chr0;
wire [39:0] asec1,asec0, amin1,amin0, ahr1,ahr0;

assign
dots={char[15],char[14],char[13],char[12],char[11],char[10],char[9],char[8]
,char[7],char[6],char[5],char[4],char[3],char[2],char[1],char[0]};

subdisplaymsg3 myfmin(clock, {1'b0,earlytime[4:0]}, fmin1, fmin0);

subdisplaymsg3 csec(clock, clocktime[5:0], csec1, csec0);
subdisplaymsg3 cmin(clock, clocktime[11:6], cmin1, cmin0);
subdisplaymsg3 chr(clock, {1'b0, clocktime[16:12]}, chr1, chr0);

subdisplaymsg3 asec(clock, alarmtime[5:0], asec1, asec0);
subdisplaymsg3 amin(clock, alarmtime[11:6], amin1, amin0);
subdisplaymsg3 ahr(clock, {1'b0, alarmtime[16:12]}, ahr1, ahr0);


always @(posedge clock) begin

    if (reset) begin
         char[15]<=displayH; char[14]<=displayA; char[13]<=displayC;
              char[12]<=displayK; char[11]<=displayspace;
char[10]<=displayT;
         char[9]<=displayH; char[8]<=displayE; char[7]<=displayspace;
         char[6]<=displayP; char[5]<=displayL; char[4]<=displayA;
         char[3]<=displayN; char[2]<=displayE; char[1]<=displayT;
         char[0]<=displayspace;  //'hack the planet'
    end
    else begin
        if (setalarm) begin
            char[15]<=displayA; char[14]<=displayL; char[13]<=displayA;
            char[12]<=displayR; char[11]<=displayM; char[10]<=displayspace;
```

```
                char[9]<=displayspace; char[8]<=displayspace;
                char[5]<=displaycolon; char[2]<=displaycolon;
                char[1] <= asec1; char[0] <= asec0;
                char[4] <= amin1; char[3] <= amin0;
                char[7] <= ahr1; char[6] <= ahr0;
        end //if setalarm
        else begin
           if (setfeature) begin
                case (featurenumber)
                    default: ;
                    3: begin        //buzzer
                             char[15]<=displayB; char[14]<=displayU;
char[13]<=displayZ;
                                  char[12]<=displayZ; char[11]<=displayE;
char[10]<=displayR;     end
                    0: begin        //light
                             char[15]<=displayL; char[14]<=displayI;
char[13]<=displayG;
                                  char[12]<=displayH; char[11]<=displayT;
char[10]<=displayspace; end
                    2: begin        //voice
                             char[15]<=displayV; char[14]<=display0;
char[13]<=displayI;
                                  char[12]<=displayC; char[11]<=displayE;
char[10]<=displayspace; end
                    1: begin       //music
                             char[15]<=displayM; char[14]<=displayU;
char[13]<=displayS;
                                  char[12]<=displayI; char[11]<=displayC;
char[10]<=displayspace; end
               endcase
                char[9] <= earlytime[5] ? displaydash : displayspace;
                char[8] <= display0;
                char[7] <= displaycolon;
                        char[6] <= fmin1; char[5] <= fmin0;
                    char[4] <= displayspace; char[3] <= displayspace;
                    char[2] <= displayspace;
                  char[1] <= displayspace; char[0] <= displayspace;
         end // if setfeature

         else //i.e. not setfeature
         begin
            char[15]<=displayC; char[14]<=displayU; char[13]<=displayR;
                 char[12]<=displayR; char[11]<=displayE;
char[10]<=displayN;
            char[9]<=displayT; char[8]<=displayspace;
```

```
            char[5]<=displaycolon; char[2]<=displaycolon;
                      char[1] <= csec1; char[0] <= csec0;
                   char[4] <= cmin1; char[3] <= cmin0;
                   char[7] <= chr1; char[6] <= chr0;
          end    //else of not setfeature
  end // else of not if setalarm
  end     // else of not reset
end   //always

endmodule
```

```
//Eleanor Foltz; Jon Spurlock;

module subdisplaymsg3(clock, dsptime, char1, char0);
input clock;
input [5:0] dsptime;
output [39:0] char1, char0;

reg [39:0] char1, char0;
reg [5:0] sectime;
parameter displayspace = 40'b00000000_00000000_00000000_00000000_00000000;
// ' '
parameter display0 = 40'b00111110_01000001_01000001_01000001_00111110; //
'0'
parameter display1 = 40'b00000000_01000010_01111111_01000000_00000000; //
'1'
parameter display2 = 40'b00100010_00110001_00101001_00100101_00100010; //
'2'
parameter display3 = 40'b01001001_01001001_01001001_01001001_00110110; //
'3'
parameter display4 = 40'b00001111_00001000_00001000_00001000_01111111; //
'4'
parameter display5 = 40'b01001111_01001001_01001001_01001001_00110001; //
'5'
parameter display6 = 40'b00111110_01001001_01001001_01001001_00110001; //
'6'
parameter display7 = 40'b00000001_00000001_00000001_00000001_01111111; //
'7'
parameter display8 = 40'b00111110_01001001_01001001_01001001_00110110; //
'8'
parameter display9 = 40'b00000110_00001001_00001001_00001001_01111110; //
'9'

always @(posedge clock) begin
       if (dsptime[5:0] < 10) begin
          char1 <= display0; sectime <= dsptime[5:0]; end
       else if (dsptime[5:0] < 20) begin
          char1 <= display1; sectime <= dsptime[5:0] - 10; end
       else if (dsptime[5:0] < 30) begin
          char1 <= display2; sectime <= dsptime[5:0] - 20; end
       else if (dsptime[5:0] < 40) begin
          char1 <= display3; sectime <= dsptime[5:0] - 30; end
       else if (dsptime[5:0] < 50) begin
          char1 <= display4; sectime <= dsptime[5:0] - 40; end
       else if (dsptime[5:0] < 60) begin
          char1 <= display5; sectime <= dsptime[5:0] - 50; end
       else begin char1 <= displayspace; sectime <= dsptime[5:0]; end
```

```
        case (sectime)
            default: char0 <= displayspace;
            0: char0 <= display0;
            1: char0 <= display1;
            2: char0 <= display2;
            3: char0 <= display3;
            4: char0 <= display4;
            5: char0 <= display5;
            6: char0 <= display6;
            7: char0 <= display7;
            8: char0 <= display8;
            9: char0 <= display9;
        endcase
end //always

endmodule
```

```
// Jon Spurlock; Eleanor Foltz; 6.111 sp05 final project

// VERSION 1.0

module light_mod (clk, rst, start, stop, max_count, intensity, active,
power);

   input clk, rst;

   input start, stop;

   input [5:0] max_count, intensity;

   output      active;

   output      power;

   reg         active;
   reg         power;

   reg [5:0]   count;

   //reg [1:0]   state, next_state;

   always @ (posedge clk)
     begin


        if (rst)
          count <= 0;
        else if (count > max_count)
          count <= 0;
        else
          count <= count + 1;

        if (rst || stop)
          begin

             active <= 0;
             power <= 0;

          end

        else if (start)
          begin
```

```
            active <=1;

            if (intensity > count)
              power <= 1;
            else
              power <= 0;

          end // if (start)

        else
          begin
            active <= 0;
            power <= 0;
          end // else: !if(start)

     end // always @ (posedge clk)

endmodule // light_mod
```

```
// Jon Spurlock; Eleanor Foltz; 6.111 sp05 final project

// VERSION 0.5
// VERSION 0.5.1 removed comma after init_status port declaration
// VERSION 0.5.2 added state initialize_1, modified initialize
// VERSION 0.5.3 added SDA_oen pass through
// VERSION 0.5.4 added readdata pass through

module music_major (clk, rst,
                    start, stop,
                    active,
                    Reset, SDA_oen, SDA, SCL,
                    //init_on, init_success, init_fail,
                    rw_read_data,
                    init_status, data_req,
                    //data_on, data_done,
                    state, rom_address
                    );

   input clk, rst;

   input start, stop;

   input data_req;
   wire data_req;

   output active;
   reg    active = 0, next_active;

   output Reset, SDA_oen, SDA, SCL;
   reg    Reset;
   wire   SDA_oen, SDA, SCL;

   //output init_start;
   reg    init_start, next_init_start;

   output [7:0] rw_read_data;
   wire [7:0] rw_read_data;

   wire   init_success, init_config, init_error;

   output [2:0] init_status;
   wire [2:0]   init_status;

   //output data_on;
   reg    data_on, next_data_on;
```

```
   //input  data_done;

   output [10:0] rom_address;

   output [2:0] state;
   reg [2:0] state, next_state;


   parameter idle = 0;
   parameter initialize = 1;
   parameter initialize_1 = 2;
   parameter data = 3;
   parameter failed_init = 4;
   parameter done = 5;

reg [9:0] music_clock;
always @ (posedge clk)
        begin
                if (rst)
                        begin
                                music_clock <= 0;
                        end
                else
                        begin
                                music_clock <= music_clock+1;
                        end
        end

   music_init_test test (.clk (music_clock[9]),
                       .rst (rst),
                       .start (init_start),
                       .busy (init_busy),
                          .SDA_oen (SDA_oen),
                       .SDA (SDA),
                       .SCL (SCL),
                          .rw_read_data (rw_read_data),
                       .init_success (init_success),
                       .init_config (init_config),
                       .init_error (init_error),
                       .data_req (data_req),
                       .rom_address (rom_address));


   assign init_status = {init_error, init_config, init_success};
```

```
   always @(posedge clk)
     begin

        if (rst || stop)
          begin
             state <= idle;
             Reset <= 0;
             init_start <= 0;
             data_on <= 0;
             active <= 0;

          end

        else if (start)
          begin
             state <= next_state;
             Reset <= 1;
             init_start <= next_init_start;
             data_on <= next_data_on;
             active <= 1;//next_active;

          end

        else
          begin
             state <= idle;
             Reset <= 0;
             init_start <= 0;
             data_on <= 0;
             active <= 0;

          end // else: !if(on)

     end // always @ (posedge clk)


   always @(state or init_start or init_busy or init_success or init_config
or init_error)// or data_done)
     begin

        next_init_start = init_start;
        next_data_on = data_on;
        next_active = active;


        case (state)
```

```
idle:
  begin
      next_state = initialize;
      next_init_start = 0;
  end

     initialize:
       begin
     if (init_busy)
        begin
                next_state = initialize_1;
            end
          else
             begin
                next_active = 1;
                next_init_start = 1;
             end

       end

     initialize_1:
       begin
           if (!init_busy)
             if (init_error)
                  begin
                     next_state = failed_init;
                  end

             else if (init_success && init_config)
                  begin
                     next_state = done;
                     //next_data_on = 1;

                  end

                  else
                  begin
                      next_state = initialize_1;
                  end

          else
                  begin
                     next_state = initialize_1;

                  end // else: !if(init_success)
```

```
            end // case: initialize_1

          data:
            begin

               if (1) //data_done)
                 begin
                    next_state = done;

                 end

               else
                 begin
                    next_state = data;
                    //next_data_on = 1;

                 end // else: !if(data_done)

            end // case: data

          failed_init:
            begin

               next_state = failed_init;

            end

          done:
            begin

               next_state = done;
               next_active = 1;

            end

        endcase // case(state)

     end // always @ (state or init_fail or init_success or data_done)

endmodule // music_major
```

```
// Jon Spurlock; Eleanor Foltz; 6.111 sp05 final project

// VERSION 0.5
// VERSION 0.5.1 fixed blocking/nonblocking in sequential always; updated
sensitivity list
// VERSION 0.5.2 added SDA_oen pass through
// VERSION 0.6.0

module music_init_test (clk, rst,
                        start, busy,
                        SDA_oen, SDA, SCL,
                        rw_read_data,
                        init_success, init_config, init_error,
                        data_req, rom_address);

   input clk, rst;

   input start;

   output busy;
   reg    busy, next_busy;

   output SDA_oen, SDA, SCL;
   wire   SDA_oen, SDA, SCL;

   output init_success, init_config, init_error;
   reg    init_success, next_init_success, init_config, next_init_config,
init_error, next_init_error;

   input data_req;

   reg    rw_start, next_rw_start;
   reg    rw_op, next_rw_op;
   reg [7:0] rw_address, next_rw_address;
   reg [7:0] rw_write_data, next_rw_write_data;
   reg [7:0] read_data, next_read_data;

   output [7:0] rw_read_data;
   wire [7:0] rw_read_data;
   wire       rw_busy, rw_success;

   reg [10:0] count, next_count;

   output [10:0] rom_address;
   reg [10:0] rom_address, next_rom_address;
   wire [8:0] roma_data, romd_data;
```

```
reg [3:0]  state, next_state;


parameter idle = 0;
parameter test = 1;
parameter test_1 = 2;
parameter test_success = 3;
parameter test_error = 4;
parameter initialize = 5;
parameter initialize_1 = 6;
parameter initialize_2 = 7;
parameter initialize_3 = 8;
parameter initialize_4 = 9;
parameter initialize_pending = 10;
parameter initialize_success = 11;
parameter initialize_error = 12;


parameter read_op = 0;
parameter write_op = 1;

parameter device_present_byte = 8'hAC;
parameter device_present_add = 8'h01;


music_rw music_rw (.clk (clk),
                   .rst (rst),
                   .start (rw_start),
                   .busy (rw_busy),
                   .r_w (rw_op),
                   .address_ext (rw_address),
                   .write_data_ext (rw_write_data),
                   .i2c_read/*_data*/ (rw_read_data),
                      .SDA_oen (SDA_oen),
                   .SDA (SDA),
                   .SCL (SCL),
                   .success (rw_success));

init_add_rom init_add_rom (rom_address, clk, roma_data);
init_data_rom init_data_rom (rom_address, clk, romd_data);

always @(posedge clk)
  begin

     if (rst)
```

```
          begin
             busy <= 0;
             rw_start <= 0;
             init_success <= 0;
             init_config <= 0;
             state <= idle;

          end

        else if (start)
          begin
             state <= next_state;
             rw_start <= next_rw_start;
             busy <= next_busy;
             rw_op <= next_rw_op;
             rw_address <= next_rw_address;
             rw_write_data <= next_rw_write_data;
             read_data <= next_read_data;
             init_success <= next_init_success;
             init_error <= next_init_error;
             count <= next_count;
             rom_address <= next_rom_address;
             init_config <= next_init_config;

          end

        else
          begin
             busy <= 0;
             rw_start <= 0;
             init_success <= 0;
             init_config <= 0;
             state <= idle;

          end // else: !if(on)

     end // always @ (posedge clk)


   always @(state or rw_start or rw_op or rw_address or rw_write_data or
rw_read_data or read_data or rw_busy or rw_success or busy or init_success
or init_error or count or rom_address or roma_data or romd_data or
data_req)
     begin

        next_rw_start = rw_start;
```

```
next_rw_op = rw_op;
next_rw_address = rw_address;
next_rw_write_data = rw_write_data;
next_read_data = read_data;
next_init_success = init_success;
next_init_error = init_error;
next_busy = busy;
next_count = count;
next_rom_address = rom_address;
next_init_config = init_config;

case (state)

  idle:
    begin

       next_init_success = 0;
       next_busy = 1;
       next_state = test;


    end // case: idle


  test:
    begin
       if (rw_busy == 1)
         begin
            next_state = test_1;
         end
       else
         begin
            next_rw_op = read_op;
            next_rw_address = device_present_add;
            next_rw_start = 1;
            next_state = test;
         end
    end // case: read


  test_1:
    begin
       if (!rw_busy)
         begin
            if (rw_success)
              begin
```

```
                next_read_data = rw_read_data;
                next_state = test_success;
              end
            else
              begin
                next_state = test_error;
              end
        end // if (!busy)
      else
        begin
          next_state = test_1;
        end // else: !if(!rw_busy)
    end // case: test_1

  test_success:
    begin
      next_rw_start = 0;
      if (read_data == device_present_byte)
        begin
          next_init_success = 1;
          next_state = initialize;
        end
      else
        begin
          next_init_error = 1;
          next_state = test_error;
        end
    end // case: test_success

  test_error:
    begin
      next_rw_start = 0;
      next_busy = 0;
      next_init_success = 0;
      next_init_error = 1;
      next_state = test_error;
    end

  initialize:
     begin
        next_count = 1;
        next_state = initialize_1;
     end

  initialize_1:
     begin
```

```
      next_rom_address = count;
      next_state = initialize_2;
   end

initialize_2:
  begin
     if (rw_busy == 1)
             begin
                next_state = initialize_3;
             end
     else
             begin
                next_rw_address = roma_data;
           next_rw_write_data = romd_data;
           next_rw_op = write_op;
           next_rw_start = 1;
           next_state = initialize_2;
             end
  end // case:

initialize_3:
  begin
     if (!rw_busy)
             begin
             next_rw_start = 0;
             if (rw_success)
               begin
                     next_state = initialize_4;
               end
             else
               begin
                     next_state = initialize_error;
               end
             end // if (!rw_busy)
     else
             begin
                next_state = initialize_3;
             end // else: !if(!rw_busy)
  end // case:

initialize_4:
   begin
      if (count == 2015)
         begin
             next_state = initialize_pending;
         end
```

```
                else
                   begin
                       next_count = count + 1;
                       next_state = initialize_1;
                   end
             end

          initialize_pending:
             begin
                if (data_req)
                   begin
                       next_state = initialize_success;
                       next_init_error = 1;
                   end
                else
                   begin
                       next_init_error = 1;
                       next_state = initialize_pending;
                   end
             end

          initialize_success:
             begin
                 next_busy = 0;
                 next_init_config = 1;
                 next_state = initialize_success;
             end

          initialize_error:
             begin
                 next_init_error = 1;
                 next_state = initialize_error;
             end

        endcase // case(state)

     end // always @ (state or rw_start or rw_op or rw_address or
rw_write_data or read_data or rw_busy or rw_success or busy)

endmodule // music_init_test
```

```
// Jon Spurlock; Eleanor Foltz; 6.111 sp05 final project

// VERSION 0.9
// VERSION 0.9.1 fixed blocking/nonblocking in sequential always; added
i2c_success to sensitivity list
// VERSION 0.9.2 added SDA_oen pass through

module music_rw  (clk, rst,
                  start, busy, //on,
                  r_w,
                  address_ext, write_data_ext, i2c_read,//_data,
                  SDA_oen, SDA, SCL,
                  success);

   input clk, rst;

   input start; //,on;

   output busy;
   reg    busy, next_busy;

   input r_w;

   input [7:0] address_ext, write_data_ext;

   reg [7:0]   address, next_address, write_data, next_write_data;

   output [7:0] i2c_read;//read_data;
   reg [7:0]    read_data, next_read_data;

   output       SDA_oen, SDA, SCL;
   wire         SDA_oen, SDA, SCL;

   output       success;
   reg          success, next_success;

   reg       i2c_start, next_i2c_start, i2c_on, next_i2c_on;
   reg [1:0] i2c_op, next_i2c_op;
   reg [7:0] i2c_write, next_i2c_write;

   wire [7:0] i2c_read;
   wire       i2c_success, i2c_busy;

   reg [4:0]  state, next_state;
```

```
parameter  idle = 0;
parameter  read = 1;
parameter  read_start1_1 = 2;
parameter  read_send1 = 3;
parameter  read_send1_1 = 4;
parameter  read_send2 = 5;
parameter  read_send2_1 = 6;
parameter  read_stop1 = 7;
parameter  read_stop1_1 = 8;
parameter  read_start2 = 9;
parameter  read_start2_1 = 10;
parameter  read_send3 = 11;
parameter  read_send3_1 = 12;
parameter  read_receive = 13;
parameter  read_receive_1 = 14;
parameter  read_stop2 = 15;
parameter  read_stop2_1 = 16;
parameter  read_success = 17;
parameter  write = 18;
parameter  write_start1_1 = 19;
parameter  write_send1 = 20;
parameter  write_send1_1 = 21;
parameter  write_send2 = 22;
parameter  write_send2_1 = 23;
parameter  write_send3 = 24;
parameter  write_send3_1 = 25;
parameter  write_stop1 = 26;
parameter  write_stop1_1 = 27;
parameter  write_success = 28;
parameter  error = 29;

parameter  rw_read = 0;
parameter  rw_write = 1;

parameter  start_op = 0;
parameter  stop_op = 1;
parameter  send_op = 2;
parameter  receive_op = 3;


parameter send_byte_1 = 8'h86;
parameter send_byte_2 = 8'h87;


always @(posedge clk)
  begin
```

```
      if (rst)
        begin
           state <= idle;
           busy <= 0;
           i2c_on <= 0;

        end

      else if (start)
        begin
           i2c_on <= 1;
           state <= next_state;
           write_data <= next_write_data;
           read_data <= next_read_data;
           address <= next_address;
           i2c_start <= next_i2c_start;
           i2c_op <= next_i2c_op;
           i2c_write <= next_i2c_write;
           busy <= next_busy;
           success <= next_success;

        end

      else
        begin
           state <= idle;
           busy <= 0;
           i2c_on <= 0;

        end // else: !if(on)

    end // always @ (posedge clk)


music_i2c i2c (.clk (clk),
               .rst (rst),
               .on (i2c_on),
               .start (i2c_start),
               .busy (i2c_busy),
               .op (i2c_op),
               .SDA_oen (SDA_oen),
               .ext_SDA (SDA),
               .SCL (SCL),
               .write_data (i2c_write),
               .receive_data (i2c_read),
```

```
                .success (i2c_success));



   always @(state or r_w or write_data_ext or address_ext or i2c_busy or
success or i2c_success)
     begin

        next_i2c_start = i2c_start;
        next_i2c_op = i2c_op;
        next_i2c_write = i2c_write;
        next_write_data = write_data;
        next_read_data = read_data;
        next_address = address;
        next_busy = busy;
        next_success = success;



        case (state)

          idle:
            begin

               if (r_w == rw_read)
                 begin
                    next_success = 0;
                    next_busy = 1;
                    next_address = address_ext;
                    next_state = read;

                 end

               else if (r_w == rw_write)
                 begin
                    next_success = 0;
                    next_busy = 1;
                    next_write_data = write_data_ext;
                    next_address = address_ext;
                    next_state = write;

                 end

               else
                 begin
                    next_busy = 0;
```

```
          next_state = idle;

        end // else: !if(r_w == write)

  end // case: idle


////////// READ ////////////////////////////////////////

read:
  begin
     if (i2c_busy == 1)
       begin
          next_i2c_start = 0;
          next_state = read_start1_1;
       end
     else
       begin
          next_i2c_op = start_op;
          next_i2c_start = 1;
          next_state = read;
       end
  end // case: read


read_start1_1:
  begin
     if (!i2c_busy)
       begin
          if (i2c_success)
            begin
               next_state = read_send1;
            end
          else
            begin
               next_state = error;
            end
       end // if (!busy)
     else
       begin
          next_state = read_start1_1;
       end // else: !if(!busy)
  end // case: read_start1_1

read_send1:
  begin
```

```
      if (i2c_busy == 1)
        begin
           next_i2c_start = 0;
           next_state = read_send1_1;
        end
      else
        begin
           next_i2c_op = send_op;
           next_i2c_write = send_byte_1;
           next_i2c_start = 1;
           next_state = read_send1;
        end // else: !if(i2c_busy == 1)
    end // case: read_send1

  read_send1_1:
    begin
       if (!i2c_busy)
         begin
            if (i2c_success)
              begin
                 next_state = read_send2;
              end
            else
              begin
                 next_state = error;
              end
         end // if (!busy)
       else
         begin
            next_state = read_send1_1;
         end // else: !if(!busy)
    end // case: read_send1_1

  read_send2:
    begin
       if (i2c_busy == 1)
         begin
            next_i2c_start = 0;
            next_state = read_send2_1;
         end
       else
         begin
            next_i2c_op = send_op;
            next_i2c_write = address;
            next_i2c_start = 1;
            next_state = read_send2;
```

```
        end // else: !if(i2c_busy == 1)
  end // case: read_send2

read_send2_1:
  begin
     if (!i2c_busy)
       begin
          if (i2c_success)
            begin
               next_state = read_stop1;
            end
          else
            begin
               next_state = error;
            end
       end // if (!busy)
     else
       begin
          next_state = read_send2_1;
       end // else: !if(!busy)
  end // case: read_send2_1

read_stop1:
  begin
     if (i2c_busy == 1)
       begin
          next_i2c_start = 0;
          next_state = read_stop1_1;
       end
     else
       begin
          next_i2c_op = stop_op;
          next_i2c_start = 1;
          next_state = read_stop1;
       end
  end // case: read_stop1


read_stop1_1:
  begin
     if (!i2c_busy)
       begin
          if (i2c_success)
            begin
               next_state = read_start2;
            end
```

```
            else
              begin
                 next_state = error;
              end
        end // if (!busy)
      else
        begin
           next_state = read_stop1_1;
        end // else: !if(!busy)
  end // case: read_stop1_1

read_start2:
  begin
     if (i2c_busy == 1)
       begin
          next_i2c_start = 0;
          next_state = read_start2_1;
       end
     else
       begin
          next_i2c_op = start_op;
          next_i2c_start = 1;
          next_state = read_start2;
       end
  end // case: read_start2


read_start2_1:
  begin
     if (!i2c_busy)
       begin
          if (i2c_success)
            begin
               next_state = read_send3;
            end
          else
            begin
               next_state = error;
            end
       end // if (!busy)
     else
       begin
          next_state = read_start2_1;
       end // else: !if(!busy)
  end // case: read_start2_1
```

```
read_send3:
  begin
     if (i2c_busy == 1)
       begin
          next_i2c_start = 0;
          next_state = read_send3_1;
       end
     else
       begin
          next_i2c_op = send_op;
          next_i2c_write = send_byte_2;
          next_i2c_start = 1;
          next_state = read_send3;
       end // else: !if(i2c_busy == 1)
  end // case: read_send3

read_send3_1:
  begin
     if (!i2c_busy)
       begin
          if (i2c_success)
            begin
               next_state = read_receive;
            end
          else
            begin
               next_state = error;
            end
       end // if (!busy)
     else
       begin
          next_state = read_send3_1;
       end // else: !if(!busy)
  end // case: read_send3_1

read_receive:
  begin
     if (i2c_busy == 1)
       begin
          next_i2c_start = 0;
          next_state = read_receive_1;
       end
     else
       begin
          next_i2c_op = receive_op;
          next_i2c_start = 1;
```

```
            next_state = read_receive;
          end // else: !if(i2c_busy == 1)
    end // case: read_receive

  read_receive_1:
    begin
       if (!i2c_busy)
         begin
            if (i2c_success)
              begin
                 next_read_data = i2c_read;
                 next_state = read_stop2;
              end
            else
              begin
                 next_state = error;
              end
         end // if (!busy)
       else
         begin
            next_state = read_receive_1;
         end // else: !if(!busy)
    end // case: read_receive_1

  read_stop2:
    begin
       if (i2c_busy == 1)
         begin
            next_i2c_start = 0;
            next_state = read_stop2_1;
         end
       else
         begin
            next_i2c_op = stop_op;
            next_i2c_start = 1;
            next_state = read_stop2;
         end
    end // case: read_stop2

  read_stop2_1:
    begin
       if (!i2c_busy)
         begin
            if (i2c_success)
              begin
                 next_state = read_success;
```

```
            end
          else
            begin
               next_state = error;
            end
       end // if (!busy)
     else
       begin
          next_state = read_stop2_1;
       end // else: !if(!busy)
  end // case: read_stop2_1

read_success:
  begin
     next_success = 1;
     next_busy = 0;
     next_state = read_success;
  end


//////// WRITE ////////////////////////////////////////

write:
  begin
     if (i2c_busy == 1)
       begin
          next_i2c_start = 0;
          next_state = write_start1_1;
       end
     else
       begin
          next_i2c_op = start_op;
          next_i2c_start = 1;
          next_state = write;
       end
  end // case: read


write_start1_1:
  begin
     if (!i2c_busy)
       begin
          if (i2c_success)
            begin
               next_state = write_send1;
            end
```

```
          else
            begin
               next_state = error;
            end
       end // if (!busy)
     else
       begin
          next_state = write_start1_1;
       end // else: !if(!busy)
  end // case: read_start1_1

write_send1:
  begin
     if (i2c_busy == 1)
       begin
          next_i2c_start = 0;
          next_state = write_send1_1;
       end
     else
       begin
          next_i2c_op = send_op;
          next_i2c_write = send_byte_1;
          next_i2c_start = 1;
          next_state = write_send1;
       end // else: !if(i2c_busy == 1)
  end // case: read_send1

write_send1_1:
  begin
     if (!i2c_busy)
       begin
          if (i2c_success)
            begin
               next_state = write_send2;
            end
          else
            begin
               next_state = error;
            end
       end // if (!busy)
     else
       begin
          next_state = write_send1_1;
       end // else: !if(!busy)
  end // case: read_send1_1
```

```
write_send2:
  begin
     if (i2c_busy == 1)
       begin
          next_i2c_start = 0;
          next_state = write_send2_1;
       end
     else
       begin
          next_i2c_op = send_op;
          next_i2c_write = address;
          next_i2c_start = 1;
          next_state = write_send2;
       end // else: !if(i2c_busy == 1)
  end // case: read_send2

write_send2_1:
  begin
     if (!i2c_busy)
       begin
          if (i2c_success)
            begin
               next_state = write_send3;
            end
          else
            begin
               next_state = error;
            end
       end // if (!busy)
     else
       begin
          next_state = write_send2_1;
       end // else: !if(!busy)
  end // case: read_send2_1

write_send3:
  begin
     if (i2c_busy == 1)
       begin
          next_i2c_start = 0;
          next_state = write_send3_1;
       end
     else
       begin
          next_i2c_op = send_op;
          next_i2c_write = write_data;
```

```
            next_i2c_start = 1;
            next_state = write_send3;
          end // else: !if(i2c_busy == 1)
    end // case: read_send2

  write_send3_1:
    begin
       if (!i2c_busy)
         begin
            if (i2c_success)
              begin
                 next_state = write_stop1;
              end
            else
              begin
                 next_state = error;
              end
         end // if (!busy)
       else
         begin
            next_state = write_send3_1;
         end // else: !if(!busy)
    end // case: read_send2_1

  write_stop1:
    begin
       if (i2c_busy == 1)
         begin
            next_i2c_start = 0;
            next_state = write_stop1_1;
         end
       else
         begin
            next_i2c_op = stop_op;
            next_i2c_start = 1;
            next_state = write_stop1;
         end
    end // case: read_stop1


  write_stop1_1:
    begin
       if (!i2c_busy)
         begin
            if (i2c_success)
              begin
```

```
                        next_state = write_success;
                     end
                   else
                     begin
                        next_state = error;
                     end
                end // if (!busy)
              else
                begin
                   next_state = write_stop1_1;
                end // else: !if(!busy)
           end // case: read_stop1_1

         write_success:
           begin
              next_success = 1;
              next_busy = 0;
              next_state = write_success;
           end


         /////////////// ERROR ////////////////

         error:
           begin
              next_success = 0;
              next_busy = 0;
              next_state = error;
           end


       endcase // case(state)

     end // always @ (state or r_w or write_data_ext or address_ext or
i2c_busy or success)

endmodule // music_r_w
```

```
// Jon Spurlock; Eleanor Foltz; 6.111 sp05 final project

// VERSION 0.9
// VERSION 0.9.1 fixed multiple 'start'
// VERSION 0.9.2 fixed ext_SDA to inout; now outputs SDA_oen for debug

module music_i2c (clk, rst,
                  on, start, busy,
                  op,
                  SDA_oen, ext_SDA, SCL,
                  write_data, receive_data,
                  success);


   input clk, rst;

   input on, start;

   output busy;
   reg    busy, next_busy;

   input [1:0] op;

   inout       ext_SDA;
   output       SDA_oen, SCL;
   reg       SDA, next_SDA, SCL, next_SCL;
   reg       SDA_oen, next_SDA_oen;

   input [7:0] write_data;

   output [7:0] receive_data;
   reg [7:0]    receive_data, next_receive_data;

   output       success;
   reg          success, next_success;

   reg [2:0]    count_i, next_count_i;

   reg [4:0]    state, next_state;


   parameter    idle = 5'd0;
   parameter    start_0 = 5'd1;
   parameter    start_1 = 5'd2;
   parameter    stop = 3;
   parameter    stop_1 = 4;
```

```
parameter    stop_2 = 5;
parameter    stop_3 = 6;
parameter    send = 7;
parameter    send_1 = 8;
parameter    send_2 = 9;
parameter    send_3 = 10;
parameter    send_ack = 11;
parameter   send_ack_charge = 22;
parameter   send_ack_0 = 23;
parameter    send_ack_1 = 12;
parameter    send_ack_2 = 13;
parameter    receive = 14;
parameter    receive_1 = 15;
parameter    receive_2 = 16;
parameter    receive_3 = 17;
parameter    receive_ack = 18;
parameter    receive_ack_1 = 19;
parameter    receive_ack_2 = 20;
parameter    error = 21;
// 22 and 23 are send_ack_charge and send_ack_0

parameter    start_op = 0;
parameter    stop_op = 1;
parameter    send_op = 2;
parameter    receive_op = 3;



assign ext_SDA = SDA_oen ? SDA : 8'hz;

always @(posedge clk)
  begin

     if (rst)
       begin
          state <= idle;
          SDA <= 1;
          SCL <= 1;
          SDA_oen <= 0;
          success <= 0;
          busy <= 0;
       end

     else if (on)
       begin
          state <= next_state;
```

```
            SDA <= next_SDA;
            SCL <= next_SCL;
            count_i <= next_count_i;
            SDA_oen <= next_SDA_oen;
            receive_data <= next_receive_data;
            success <= next_success;
            busy <= next_busy;
          end

        else
          begin
            state <= idle;
            SDA <= 1;
            SCL <= 1;
            SDA_oen <= 0;
            success <= 0;
            busy <= 0;

          end // else: !if(on)

      end // always @ (posedge clk)


    always @(state or start or op or SDA or ext_SDA or SCL or count_i)
      begin

        next_SDA = SDA;
        next_SCL = SCL;
        next_count_i = count_i;
        next_SDA_oen = SDA_oen;
        next_receive_data = receive_data;
        next_success = success;
        next_busy = busy;


        case (state)

          idle:
            begin

              if (start)
                begin
                next_busy = 1;
                next_success = 0;
                if (op == start_op)
                  begin
```

```
            next_SDA_oen = 1;
            next_state = start_0;
          end
        else if (op == stop_op)
          begin
            //next_SDA_oen = 1;
            next_state = stop;
          end
        else if (op == receive_op)
          begin
            next_state = receive;
          end
        else if (op == send_op)
          begin
            //next_SDA_oen = 1;
            next_state = send;
          end
        else
          begin
            next_state = idle;
          end
        end // if (start)

      else
        begin
           next_SDA_oen = 0;
           next_state = idle;
           next_busy = 0;
        end // else: !if(start)

  end // case: idle

start_0:
  begin

      if (SDA && SCL)
        begin
           next_state = start_1;
           next_SDA_oen = 1;
           next_SDA = 0;
        end
      else
        next_state = error;

  end // case: start
```

```
start_1:
  begin
     next_SCL = 0;
     next_success = 1;
     next_state = idle;
  end


stop:
  begin

     if (!SCL)
       begin
          next_SDA_oen = 1;
          next_state = stop_1;
       end
     else
       next_state = error;

  end // case: stop

stop_1:
  begin
     next_state = stop_2;
     next_SDA = 0;
  end

stop_2:
  begin
     next_state = stop_3;
     next_SCL = 1;
  end

stop_3:
  begin
     next_state = idle;
     next_SDA = 1;
     next_success = 1;
     next_state = idle;
  end


send:
  begin

     if (!SCL)
```

```
        begin
           next_SDA_oen = 1;
           next_state = send_1;
           next_count_i = 7;
        end
      else
        next_state = error;

  end // case: receive

send_1:
  begin
     case (count_i)
       7: next_SDA = write_data[7];
       6: next_SDA = write_data[6];
       5: next_SDA = write_data[5];
       4: next_SDA = write_data[4];
       3: next_SDA = write_data[3];
       2: next_SDA = write_data[2];
       1: next_SDA = write_data[1];
       0: next_SDA = write_data[0];
     endcase // case(count_i)
     next_state = send_2;
  end

send_2:
  begin
     next_SCL = 1;
     next_state = send_3;
  end

send_3:
  begin
     next_SCL = 0;
     if (count_i == 0)
       begin
          next_state = send_ack;
       end
     else
       begin
          next_state = send_1;
          next_count_i = count_i - 1;
       end
  end // case: send_3

send_ack:
```

```
      begin
              next_SDA = 1;
              next_state = send_ack_charge;
      end

send_ack_charge:
      begin
              next_SDA = 1;
              next_state = send_ack_0;
      end

send_ack_0:
  begin
     next_SDA_oen = 0;
     next_state = send_ack_1;
  end

send_ack_1:
  begin
     next_SCL = 1;
     next_state = send_ack_2;
  end

send_ack_2:
  begin
     if (!ext_SDA)
       begin
          next_SCL = 0;
          next_success = 1;
          next_state = idle; //send_ack_3;
       end
     else
       next_state = error;
  end


receive:
  begin

     if (!SCL)
       begin
          next_count_i = 7;
          next_SDA_oen = 0;
          next_state = receive_1;
       end
     else
```

```
        next_state = error;

    end // case: receive

  receive_1:
    begin
       next_SCL = 1;
       next_state = receive_2;
    end

  receive_2:
    begin
       case (count_i)
         7: next_receive_data[7] = ext_SDA;
         6: next_receive_data[6] = ext_SDA;
         5: next_receive_data[5] = ext_SDA;
         4: next_receive_data[4] = ext_SDA;
         3: next_receive_data[3] = ext_SDA;
         2: next_receive_data[2] = ext_SDA;
         1: next_receive_data[1] = ext_SDA;
         0: next_receive_data[0] = ext_SDA;
       endcase // case(count_i)
       next_state = receive_3;
    end // case: receive_2

  receive_3:
    begin
       next_SCL = 0;
       if (count_i == 0)
         begin
            next_state = receive_ack;
         end
       else
         begin
            next_count_i = count_i - 1;
            next_state = receive_1;
         end
    end // case: receive_3

  receive_ack:
    begin
       next_SDA_oen = 1;
       next_SDA = 0;
       next_state = receive_ack_1;
    end
```

```
            receive_ack_1:
              begin
                 next_SCL = 0;
                 next_state = receive_ack_2;
              end

            receive_ack_2:
              begin
                 next_SCL = 0;
                 next_success = 1;
                 next_state = idle;
              end

            error:
              begin
                 next_busy = 0;
                 next_success = 0;
                 next_state = error;
              end


         endcase // case(state)

      end // always @ (state or start or op or SDA or SCL or count_i)

endmodule // music_i2c
```

```
//Eleanor Foltz; Jon Spurlock


module buzzermod(reset, clock_27mhz,
                 start, stop,
                 busy, sound);

input reset, clock_27mhz, start, stop;
output busy;
output [7:0] sound;

reg [7:0] sound;
reg [7:0] dividercount;
reg busy;
//outputs an 8bit sawtooth wave at a
// frequency 27mHz/256*(241) = 443Hz

//The sound output will need to be registered
//   before being sent to the D/A converter.
//   That registering is not done here.
//   Decision made 5/6/05 EF

parameter RESET = 0; parameter IDLE = 1;
parameter INIT = 2; parameter COUNTING = 3;
parameter INCR = 4;
reg [2:0] state;

always @(posedge clock_27mhz) begin
    if (reset) begin
        sound <= 8'b0;
        state <= RESET; end
    else begin
        case (state)
            default: state <=0;
            RESET: begin
                dividercount <= 8'b0;
                sound <= 8'b0 ;
                busy <= 0;
                state <= IDLE; end
            IDLE: state <= start ? INIT : IDLE;
            INIT: begin
                sound <= 8'b0;
                busy <= 1;
                state <= COUNTING; end
            COUNTING: begin               //below was 240
                state <= stop? RESET:(dividercount == 240) ? INCR:
```

```
COUNTING;
                dividercount <= dividercount + 1; end
            INCR: begin
                sound <= sound + 1;
                dividercount <= 8'b0;
                state <= stop? RESET:COUNTING; end
        endcase
    end // else for !reset
end //always
endmodule


//OLD CODE (REVISION MADE 5/6)
//always @(posedge clock_27mhz) begin
//  if (reset) begin
//    dividercount = 0;
//    sound = 0;
//  end
//  else begin
//      if (buzzerGO) begin
//        dividercount = dividercount + 1;
//        if (dividercount == 240) begin
//            sound = sound + 1;
//        end end
//      else begin
//        sound = 0;
//      end // else for if gobuzzer
//  end //else for if reset
//end //always
```

```
//Eleanor Foltz; Jon Spurlock;

module voicemod(reset, clock_27mhz,
                start, stop, record,
                status, soundin, CSbarAD, RorWbar,
                soundout, busy, state,addr);

input reset, clock_27mhz, start, stop, record, status;
input [7:0] soundin;
output busy, CSbarAD, RorWbar;
output [7:0] soundout;
output [3:0] state;
output [16:0] addr;

reg [7:0] soundout;
reg [10:0] rcnt, internalcnt;
reg busy,CSbarAD, RorWbar;
reg [16:0] SRAMaddr;

parameter RESET = 0; parameter IDLE = 1;
parameter INITPLAY = 2; parameter INITRECORD = 3;
parameter PLAY1 = 4; parameter PLAY2 = 5;
parameter WAITPLAY = 6; parameter WAITRECORD = 7;
parameter CONVERSIONWAIT = 8;parameter RECORD = 9;

reg [3:0] state;
reg SRAMWriteEnable;
reg [7:0] SRAMdatain;
wire [7:0] SRAMout;
//assign rcnttop = rcnt [3:0];
assign addr = SRAMaddr;
ram8_1350 myRAM(clock_27mhz, 1'b1, SRAMWriteEnable, SRAMaddr, SRAMdatain,
                     SRAMout);

always @(posedge clock_27mhz) begin
    if (reset) state <= RESET;
    else begin
        case (state)
            default: state <= RESET;
            RESET: begin
                state <= IDLE; busy <= 0; soundout<=8'b0; end
            IDLE: begin
                busy <= 0; soundout<=8'b0;
                state <= start ? INITPLAY : record ? INITRECORD : IDLE;
            end
            INITPLAY: begin
```

```
                SRAMWriteEnable <= 0;
                rcnt <= 1348;
                SRAMaddr <= 0;
                busy <= 1;
                state <= WAITPLAY; end
            WAITPLAY: begin
                state <= stop? IDLE : (rcnt ==1350) ? PLAY1 : WAITPLAY;
                rcnt <= rcnt + 1;
                internalcnt <= 0;
            end
            PLAY1: begin
                rcnt <= 0;
                internalcnt <= internalcnt + 1;
                //is there a wait time to access info in the SRAM?
                state <= (internalcnt == 3) ? PLAY2 : PLAY1;
            end
            PLAY2: begin
                soundout <= SRAMout;
                SRAMaddr <= SRAMaddr + 1;
                state <= (SRAMaddr == 100000) ? IDLE: WAITPLAY;
            end
            INITRECORD: begin
                busy <= 1;
                state <= RECORD;
                rcnt <= 17; //was 13
                SRAMaddr <= 0; end
            WAITRECORD: begin
                rcnt <= rcnt + 1;  //below s/b 100000
                state <= (SRAMaddr == 100000)? IDLE: ((rcnt ==1350) ?
CONVERSIONWAIT: WAITRECORD);
            end
            CONVERSIONWAIT: begin
                rcnt <= 0;
                state <= status ? RECORD : CONVERSIONWAIT; end
            RECORD: begin
                rcnt <= rcnt + 1;
                case (rcnt)
                    default: ;//original were 0,12,13,14,28,29
                    0: begin RorWbar <= 1; CSbarAD <= 0; end
                    8: begin CSbarAD <= 1;
                        SRAMdatain <= soundin; end
                    13: begin
                        SRAMWriteEnable <=1; end
                    15: begin
                        SRAMWriteEnable <=0; end
                    17: begin RorWbar <= 0; CSbarAD <= 0; end
```

```
                    27: begin RorWbar <= 1; CSbarAD <= 1; end
                    29: begin
                        SRAMaddr <= SRAMaddr +1;
                        state <= WAITRECORD; end
                endcase
            end
        endcase //on state
    end //else of !reset
end // always

endmodule
```

```
//credit to webpage...

module ram8_1350 (clock, en, we, addr, wdata, rdata);

  input clock;
  input en;            // device enable: 1=enabled, 0=disabled
  input we;            // write enable: 1=write, 0=read
  input [16:0] addr;    // address
  output [7:0] rdata; // read data port
  input [7:0] wdata;  // write data port

  reg [7:0] memory[1351:0];
  reg [7:0] rdata;
  always @(posedge clock)
    if (en)
      begin
        if (we)
          memory[addr] <= wdata;
        rdata <= memory[addr];
      end

endmodule
```

```
//Eleanor Foltz; Jon Spurlock;

//version 2.0
// this version creates a constant audioout signal for
//   entire 1350 cycles of the 27mhz clock.

module audio (rst, clk, alarmon, buzzersound, voicesound,
                                  audioout, CSbarDA);
input rst, clk, alarmon;
input [7:0] buzzersound, voicesound;

output [7:0] audioout;
output CSbarDA;
reg CSbarDA;
wire [8:0] sumation;
reg [7:0] audioout;
reg [10:0] count;

assign sumation = buzzersound + voicesound;

always @(posedge clk) begin
    CSbarDA <= ~alarmon;  //CSbar is active low
    if (rst) begin
        audioout <= sumation[8:1];
        count <= 0; end
    else if (count == 1350) begin
       count <= 0;
       audioout <= sumation[8:1]; end
    else count <= count + 1;
end //always


endmodule
```