Final Project Report (Appendix):

Wireless Musical Electrocardiogram

Amy Tang and Sinit Vitavasiri

(Group 12)

TA: Hyunjoo Jenny Lee

Massachusetts Institute of Technology

Department of Electrical Engineering and Computer Science

6.111 – Introductory Digital Systems Laboratory

Appendix A: Switch Assignments

switch[4]	switch[3]	switch[2]	switch[1]	switch[0]	Function
X	X	X	X	Up	Enable EKG input (Testing Mode)
X	X	X	X	Down	Disable EKG input (Testing Mode)
X	X	X	Up	X	Enable Overall System
X	X	X	Down	X	Disable Overall System
X	Down	Down	Up	X	Disable Heart Rhythm Sound and Abnormality Detection Alarm*
X	Down	Up	Up	X	Enable Abnormality Detection Alarm*
X	Up	Down	Up	X	Enable Heart Rhythm Sound*
X	Up	Up	Up	X	Enable Heart Rhythm Sound and Abnormality Detection Alarm*
Up	X	X	Up	X	Enable Classical Music* (Heart-Rate-Controlled Music Mode)

* The volume can be adjusted over the range of 0-15 using the up and down pushbuttons.

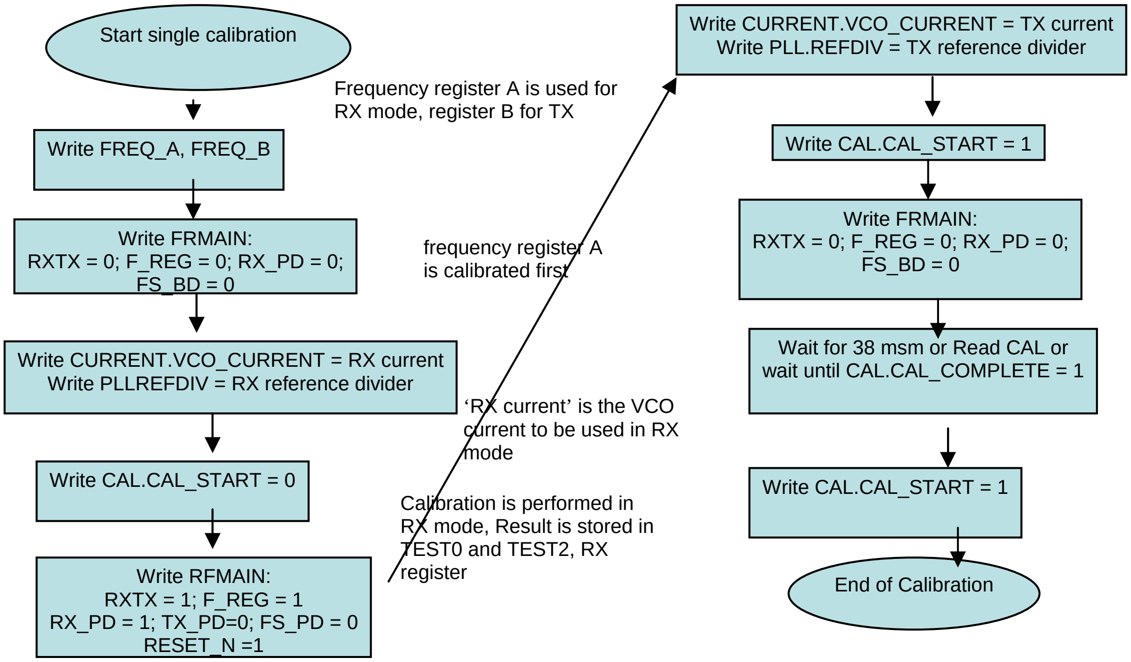
switch[7]	Function
Up	Heart-Rate Display
Down	Volume Display

switch[5]	Function
Up	Enable Wireless Transmission
Down	Disable Wireless Transmission

Appendix B: Tempo-Changing Algorithm (Matlab Code)

```
function output = timescale(sig, compression, maxfreq)
% takes in a signal in the time domain and scales its length,
% thus increasing its tempo. It scales the signal by compression,
% where compression is less than 1.
% It takes in maxfreq in order to compute how often to remove samples.

if nargin < 3, maxfreq = 4096; end
n = length(sig);

% Computes how often to remove samples
Timediv = floor(.08*maxfreq*2)

% Computes how many samples to remove
remove = floor((1-compression)*timediv)
output = 0;

% Remove samples, and recombine signals
for i = remove+1:(timediv±remove):(n-timediv)
   output = [output; sig((i-remove):(i+timediv-remove))];
end
```

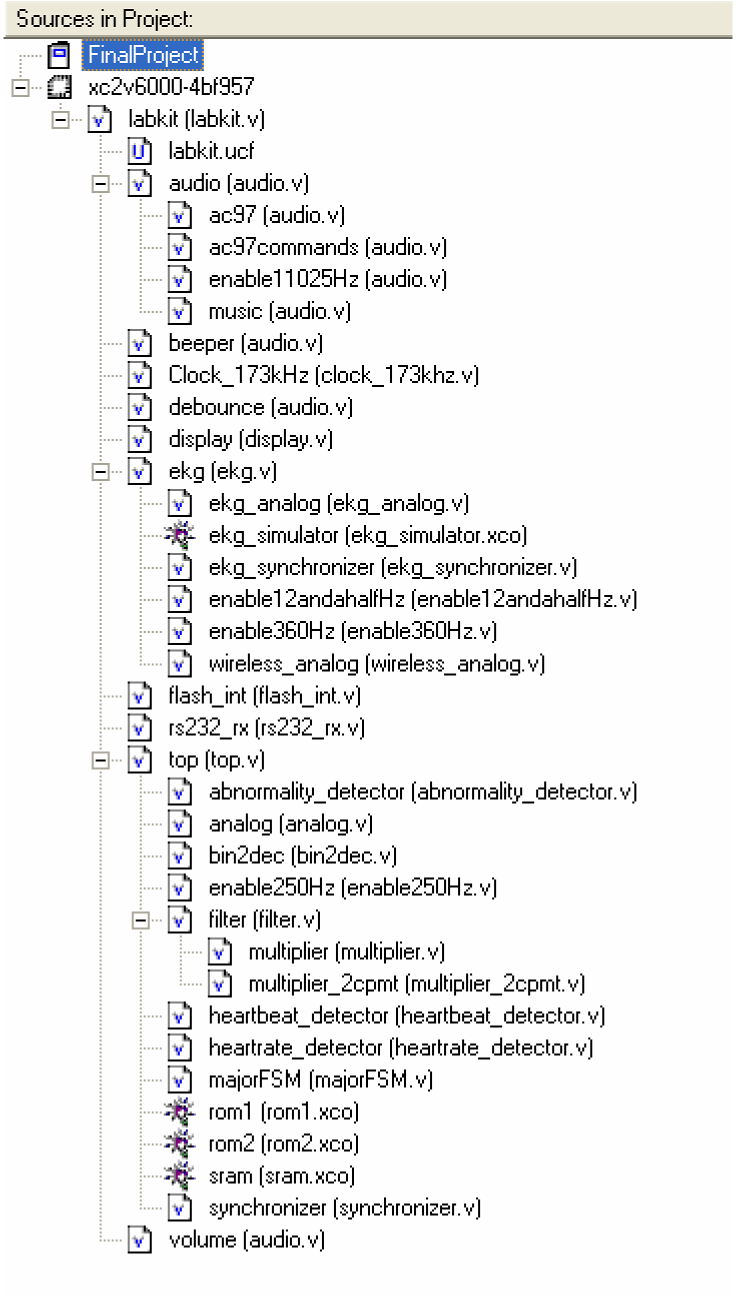
Appendix C: Wireless Calibration Algorithm


Start single calibration
Frequency register A is used for RX mode, register B for TX
Write FREQ_A, FREQ_B
Write FRMAIN: RXTX = 0; F_REG = 0; RX_PD = 0; FS_BD = 0
frequency register A is calibrated first
Write CURRENT.VCO_CURRENT = RX current Write PLLREFDIV = RX reference divider
'RX current' is the VCO current to be used in RX mode
Write CAL.CAL_START = 0
Calibration is performed in RX mode, Result is stored in TEST0 and TEST2, RX register
Write RFMAIN: RXTX = 1; F_REG = 1 RX_PD = 1; TX_PD=0; FS_PD = 0 RESET_N =1
Write CURRENT.VCO_CURRENT = TX current Write PLL.REFDIV = TX reference divider
Write CAL.CAL_START = 1
Write FRMAIN: RXTX = 0; F_REG = 0; RX_PD = 0; FS_BD = 0
Wait for 38 msm or Read CAL or wait until CAL.CAL_COMPLETE = 1
Write CAL.CAL_START = 1
End of Calibration

Control Flow of a Calibration Instantiation of a Transceiver (Transmittance and Receiving) System

Appendix D: Verilog/C Codes

All the codes are sorted alphabetically. Below is the hierarchy of the overall system.



Sources in Project:

```
FinalProject
xc2v6000-4bf957
  labkit (labkit.v)
    labkit.ucf
    audio (audio.v)
      ac97 (audio.v)
      ac97commands (audio.v)
      enable11025Hz (audio.v)
      music (audio.v)
    beeper (audio.v)
    Clock_173kHz (clock_173khz.v)
    debounce (audio.v)
    display (display.v)
    ekg (ekg.v)
      ekg_analog (ekg_analog.v)
      ekg_simulator (ekg_simulator.xco)
      ekg_synchronizer (ekg_synchronizer.v)
      enable12andahalfHz (enable12andahalfHz.v)
      enable360Hz (enable360Hz.v)
      wireless_analog (wireless_analog.v)
    flash_int (flash_int.v)
    rs232_rx (rs232_rx.v)
    top (top.v)
      abnormality_detector (abnormality_detector.v)
      analog (analog.v)
      bin2dec (bin2dec.v)
      enable250Hz (enable250Hz.v)
      filter (filter.v)
        multiplier (multiplier.v)
        multiplier_2cpmt (multiplier_2cpmt.v)
      heartbeat_detector (heartbeat_detector.v)
      heartrate_detector (heartrate_detector.v)
      majorFSM (majorFSM.v)
      rom1 (rom1.xco)
      rom2 (rom2.xco)
      sram (sram.xco)
      synchronizer (synchronizer.v)
    volume (audio.v)
```

There are other five files that are used to program the Flash ROM, namely: *flashtest*, *romsound*, *test_fsm_no_erase*, *test_fsm_old*, and *test_fsm_read_ROM*. The C codes used to implement the wireless system can be found in *wireless1*, *wireless2*, and *wireless3*.

```verilog
module abnormality_detector(clk, reset_sync, heartrate, abnormal);

input clk, reset_sync;
input[8:0] heartrate;
output abnormal;

wire abnormal;
reg[21:0] count;

always @ (posedge clk)
begin
   count <= count + 1;
end

assign abnormal = (heartrate <= 9'd50 || (heartrate >= 9'd200 && heartrate != 9'b111111111)) ? count[21] : 1'b0;

endmodule
```

```verilog
module analog(clk, reset_sync, ADC_start, DAC_start, status_sync, int_data, analog_busy, r_wbar, cs_bar, cs_bar_DA, ext_data, read_data);

input clk, reset_sync, ADC_start, DAC_start, status_sync;
input[7:0] int_data;
output analog_busy, r_wbar, cs_bar, cs_bar_DA;
output[7:0] read_data;
inout[7:0] ext_data;
wire[7:0] ext_data;

reg[3:0] state, nextstate;
reg r_wbar_int, r_wbar;
reg cs_bar_int, cs_bar;
reg cs_bar_DA_int, cs_bar_DA;
reg LE_ADC, LE_DAC, LE_DAC_int;
reg analog_busy, analog_busy_int;
reg[7:0] read_data;

assign ext_data = LE_DAC ? int_data : 8'hz;

// State declarations
parameter IDLE = 0;
parameter DAC0 = 1;
parameter DAC1 = 2;
parameter DAC2 = 3;
parameter DAC3 = 4;
parameter WRITE0 = 5;
parameter WRITE1 = 6;
parameter WRITE2 = 7;
parameter WAITSTATUSHIGH = 8;
parameter WAITSTATUSLOW = 9;
parameter READ0 = 10;
parameter READ1 = 11;
parameter READ2 = 12;
parameter READ3 = 13;

always @ (posedge clk)
begin
    if (!reset_sync) state <= IDLE;
    else state <= nextstate;

    r_wbar <= r_wbar_int;
    cs_bar <= cs_bar_int;
    cs_bar_DA <= cs_bar_DA_int;
    LE_DAC <= LE_DAC_int;
    analog_busy <= analog_busy_int;

    if (LE_ADC) read_data <= ext_data;
end

always @ (state or status_sync or DAC_start or ADC_start)
begin
    r_wbar_int = 1;
    cs_bar_int = 1;
    cs_bar_DA_int = 1;
    LE_ADC = 0;
    LE_DAC_int = 0;
    analog_busy_int = 1;

    case (state)
      IDLE:    begin
                   analog_busy_int = 0;
                   if (DAC_start) nextstate = DAC0;
                   else if (ADC_start) nextstate = WRITE0;
                     else nextstate = IDLE;
                 end

        DAC0:    begin
                     LE_DAC_int = 1;
                     nextstate = DAC1;
                 end

        DAC1:    begin
                     LE_DAC_int = 1;
                     cs_bar_DA_int = 0;
                     nextstate = DAC2;
                 end

        DAC2:    begin
```

```
                     LE_DAC_int = 1;
                     nextstate = DAC3;
                 end

        DAC3:    begin
                     LE_DAC_int = 0;
                     nextstate = IDLE;
                 end

        WRITE0:  begin
                     r_wbar_int = 0;
                     cs_bar_int = 0;
                     nextstate = WRITE1;
                 end

        WRITE1:  begin
                     r_wbar_int = 0;
                     cs_bar_int = 0;
                     nextstate = WRITE2;
                 end

        WRITE2:  begin
                     r_wbar_int = 0;
                     cs_bar_int = 0;
                     nextstate = WAITSTATUSHIGH;
                 end

        WAITSTATUSHIGH:
                 begin
                     cs_bar_int = 0;
                     if (status_sync) nextstate = WAITSTATUSLOW;
                     else nextstate = WAITSTATUSHIGH;
                 end

        WAITSTATUSLOW:
                 begin
                     cs_bar_int = 0;
                     if (!status_sync) nextstate = READ0;
                     else nextstate = WAITSTATUSLOW;
                 end

        READ0:   begin
                     cs_bar_int = 0;
                     nextstate = READ1;
                 end

        READ1:   begin
                     cs_bar_int = 0;
                     nextstate = READ2;
                 end

        READ2:   begin
                     cs_bar_int = 0;
                     LE_ADC = 1;
                     nextstate = READ3;
                 end

        READ3:   begin
                     nextstate = IDLE;
                 end

        default: nextstate = IDLE;
    endcase
end

endmodule
```

```
module audio (reset, clock_27mhz, audio_reset_b, ac97_sdata_out, ac97_sdata_in,
         ac97_synch, ac97_bit_clock, volume, mode, flash_data, flash_address, heartrate);

   input reset, clock_27mhz;
   output audio_reset_b;
   output ac97_sdata_out;
   input ac97_sdata_in;
   output ac97_synch;
   input ac97_bit_clock;
   input [4:0] volume;
   input mode;
   input [15:0] flash_data;
   output [22:0] flash_address;
   input [8:0] heartrate;

   wire ready;
   wire [7:0] command_address;
   wire [15:0] command_data;
   wire command_valid;
   reg [19:0] left_out_data, right_out_data;
   wire [19:0] left_in_data, right_in_data, music_data;
   wire [15:0] flash_data;
   wire [22:0] flash_address;
   wire enable11025Hz;

   ///////////////////////////////////////////////////////////////////////////
   //
   // Reset Controller
   //
   ///////////////////////////////////////////////////////////////////////////

   reg audio_reset_b;
   reg [9:0] reset_count;

   always @(posedge clock_27mhz) begin
      if (reset)
   begin
      audio_reset_b = 1'b0;
      reset_count = 0;
   end
      else if (reset_count == 1023)
   audio_reset_b = 1'b1;
      else
   reset_count = reset_count+1;
   end

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

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

   enable11025Hz enable11025Hz1(clock_27mhz, reset, enable11025Hz);

   music music1(clock_27mhz, enable11025Hz, music_data, mode, flash_data, flash_address, heartrate);

   always @(mode or left_in_data or right_in_data or music_data)
     case (mode)
       1'b0: begin
     left_out_data = 20'h00000;
     right_out_data = 20'h00000;
       end
       1'b1: begin
     left_out_data = music_data;
     right_out_data = music_data;
       end
     endcase

endmodule

   ///////////////////////////////////////////////////////////////////////////
   //
   // Beeper
   //
   ///////////////////////////////////////////////////////////////////////////

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

```
   input reset, clock_27mhz, enable;
   output beep;

   reg [15:0] count;
   reg clock_1khz;

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

   assign beep = enable && clock_1khz;

endmodule

   ///////////////////////////////////////////////////////////////////////////////
   //
   // Volume Controller
   //
   ///////////////////////////////////////////////////////////////////////////////

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

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

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

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

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

endmodule

   ///////////////////////////////////////////////////////////////////////////////
   //
   // AC97
   //
   ///////////////////////////////////////////////////////////////////////////////

module ac97 (ready,
        command_address, command_data, command_valid,
        left_data, left_valid,
        right_data, right_valid,
        left_in_data, right_in_data,
        ac97_sdata_out, ac97_sdata_in, ac97_synch, ac97_bit_clock);

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

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

   reg ready;

   reg ac97_sdata_out;
   reg ac97_synch;

   reg [7:0] bit_count;

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

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

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

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

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

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

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

      if ((bit_count >= 0) && (bit_count <= 15))
   // Slot 0: Tags
   case (bit_count[3:0])
     4'h0: ac97_sdata_out <= 1'b1;      // Frame valid
     4'h1: ac97_sdata_out <= l_cmd_v;   // Command address valid
     4'h2: ac97_sdata_out <= l_cmd_v;   // Command data valid
     4'h3: ac97_sdata_out <= l_left_v;  // Left data valid
     4'h4: ac97_sdata_out <= l_right_v; // Right data valid
     default: ac97_sdata_out <= 1'b0;
   endcase

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

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

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

      bit_count <= bit_count+1;

   end // always @ (posedge ac97_bit_clock)

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

endmodule

   ///////////////////////////////////////////////////////////////////////////////
   //
   // AC97commands
   //
   ///////////////////////////////////////////////////////////////////////////////

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

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

   reg [23:0] command;
   reg command_valid;

   reg old_ready;
   reg done;
   reg [3:0] state;

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

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

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

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

      case (state)
   4'h0: // Read ID
     begin
     command <= 24'h80_0000;
     command_valid <= 1'b1;
     end
     4'h1: // Read ID
     command <= 24'h80_0000;
   4'h2: // enable changing sampling rate
     command <= 24'h2A_0001;
//********** Sampling Rate **********************************************//
   4'h3: // Sampling rate
     command <= 24'h2C_2B11;
   4'h4: // Sampling rate
     command <= 24'h32_2B11;
   4'h5: // Master volume
     command <= { 8'h02, 3'b000, vol, 3'b000, vol };
   4'h6: // Aux volume
     command <= { 8'h04, 3'b000, vol, 3'b000, vol };
   4'h7: // Mono volume
     command <= 24'h06_8000;
   4'h8: // PCM volume
     command <= 24'h18_0808;
   4'h9: // Record source select
     //if (source)
     //  command <= 24'h1A_0000; // microphone
     //else
     command <= 24'h1A_0404; // line-in
   4'hA: // Record gain
     command <= 24'h1C_0000;
   4'hB: // Line in gain
     command <= 24'h10_8000;
   4'hC: // Set beep volume
     command <= 24'h0A_0000;
   //4'hF: // Misc control bits
     //command <= 24'h76_8000;
   default:
     command <= 24'h80_0000;
      endcase // case(state)

      old_ready <= ready;

   end // always @ (posedge clock)

endmodule // ac97commands

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

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

   input reset, clock, noisy;
   output clean;

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

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

endmodule


////////////////////////////////////////////////////////////////////////////////
//
// Music Module
//
////////////////////////////////////////////////////////////////////////////////

module music(clock, ready, pcm_data, start, flash_data, flash_address, heartrate);

   input clock;
   input ready;
   input start;
   output [19:0] pcm_data;
   input [15:0] flash_data;
   output [22:0] flash_address;
   input [8:0] heartrate;

   reg rdy, old_ready;
   reg [19:0] pcm_data;
   reg [22:0] flash_address;
   reg [10:0] count;
   reg [22:0] remove;

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

   always @(posedge clock)
   begin
    if (heartrate == 9'b111111111 || heartrate < 9'd80) remove <= 1;
    else if (heartrate < 9'd100) remove <= 23'd352;
    else if (heartrate < 9'd120) remove <= 23'd705;
    else if (heartrate < 9'd140) remove <= 23'd882;
    else remove <= 23'd1058;

      if (!start) begin
         old_ready <= 1'b0;
         pcm_data <= 20'h00000;
      flash_address <= 0;
      count <= 0;
      end
    else begin
      if (rdy && ~old_ready) begin
         pcm_data <= {flash_data[7:0], 12'h000};
         if (count > 11'd1764) begin
            count <= 1;
            if (flash_address > 23'h00FFFF-23'd1764-remove) flash_address <= 0;
            else flash_address <= flash_address + remove;
            end
         else begin
            count <= count + 1;
            if (flash_address == 23'h00FFFF) flash_address <= 0;
            else flash_address <= flash_address+1;
            end
         end
         old_ready <= rdy;
         rdy <= ready;
      end
   end

```

```
endmodule


////////////////////////////////////////////////////////////////////////////////
//
// Enable 11.025 kHz
//
////////////////////////////////////////////////////////////////////////////////

module enable11025Hz(clk, reset, enable11025Hz);

input clk, reset;
output enable11025Hz;
reg enable11025Hz;
reg[14:0] counter;

always @ (posedge clk)
begin
   if (reset)
   begin
      enable11025Hz <= 1'b0;
      counter <= 15'd0;
   end
   else if(counter == 16'd2448)      // 27 MHz --> 11.025 kHz
   begin
      enable11025Hz <= 1'b1;
      counter <= 15'd0;
   end
   else
   begin
      enable11025Hz <= 1'b0;
      counter <= counter + 1;
   end
end

endmodule
```

```
module bin2dec(clk, reset_sync, convert, heartrate, digit2, digit1, digit0);

input clk, reset_sync, convert;
input[8:0] heartrate;
output[3:0] digit2, digit1, digit0;

reg[8:0] count;
reg[3:0] digit2, digit1, digit0;
reg[3:0] digit2_int, digit1_int, digit0_int;
reg[1:0] state, nextstate;

parameter IDLE = 2'b00;
parameter WAIT = 2'b01;
parameter DO_CONVERT = 2'b10;
parameter DO_CONVERT2 = 2'b11;

always @ (posedge clk)
begin
   if (!reset_sync) state <= IDLE;
   else state <= nextstate;

   if (state == IDLE) begin
      digit2 <= 4'b1111;
      digit1 <= 4'b1111;
      digit0 <= 4'b1111;
      count <= 0;
      end
   else if (state == WAIT) begin
      count <= 0;
      digit2_int <= 0;
      digit1_int <= 0;
      digit0_int <= 0;
      end
   else if (state == DO_CONVERT) begin
      if (count == heartrate) begin
         digit2 <= digit2_int;
         digit1 <= digit1_int;
         digit0 <= digit0_int;
         end
      else if (digit1_int == 4'b1001 && digit0_int == 4'b1001) begin
         digit2_int <= digit2_int + 1;
         digit1_int <= 0;
         digit0_int <= 0;
         end
      else if (digit0_int == 4'b1001) begin
         digit1_int <= digit1_int + 1;
         digit0_int <= 0;
         end
      else  begin
         digit0_int <= digit0_int + 1;
         end
      end
   else  if (state == DO_CONVERT2)
      count <= count + 1;

end

always @ (state or convert)
begin
   case (state)
      IDLE: begin
               nextstate = WAIT;
            end

      WAIT:       begin
               if (convert) nextstate = DO_CONVERT;
                        else nextstate = WAIT;
                  end

      DO_CONVERT:  begin
               if (count == heartrate) nextstate = WAIT;
               else  nextstate = DO_CONVERT2;
            end

      DO_CONVERT2: begin
               nextstate = DO_CONVERT;
            end

         default: nextstate = IDLE;
      endcase
end

endmodule
```

```
// Clock of 173 kHz //

// Uses 27MHz clock input to generate 173kHz Clock
// The 173kHz clock is used for the different states in
// FSM

module Clock_173kHz (clk, reset, enable);

   input clk, reset;
   output enable;

   reg enable;
      reg [20:0] count;

   always @ (posedge clk)
   begin
      if (!reset)
         begin
            count <= 21'd0;
            enable <= 0;
         end
      else if (count == 21'd156)    // 27000000/156
         begin
            enable <= 1;
            count <= 21'd0;
         end
      else
         begin
            enable <= 0;
            count <= count + 1;
         end
   end

endmodule
```

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

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

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

   reg disp_data_out, disp_rs, disp_ce_b, disp_reset_b;

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

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

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

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

   assign dreset = (reset_count != 0);

   assign disp_clock = ~clock;

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

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

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

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

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

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

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

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

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

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

endmodule

```

```
module ekg(clk, reset, cs_bar_DA_ekg, data, rxdata, wireless_dataout, cs_bar_DA_wireless);

   input clk, reset;
   output cs_bar_DA_ekg, cs_bar_DA_wireless;
   output[7:0] data, wireless_dataout;
   input[7:0] rxdata;

   wire reset_sync, sample_ekg, sample_wireless;
   wire[11:0] ekg_addr;
   wire[7:0] ekg_q;

   ekg_synchronizer ekg_synchronizer1(clk, reset, reset_sync);
   enable360Hz enable360Hz1(clk, reset_sync, sample_ekg);
   ekg_simulator ekg_simulator1(ekg_addr, ekg_q);
   ekg_analog ekg_analog1(clk, reset_sync, sample_ekg, cs_bar_DA_ekg, ekg_addr, ekg_q, data);
   wireless_analog wireless_analog1(clk, reset_sync, sample_wireless, cs_bar_DA_wireless, rxdata, wireless_dataout);
   enable12andahalfHz enable12andahalfHz1(clk, reset_sync, sample_wireless);

endmodule
```

```
module ekg_analog(clk, reset_sync, sample_ekg, cs_bar_DA_ekg, count, ekg_q, data);

input clk, reset_sync, sample_ekg;
input[7:0] ekg_q;
output cs_bar_DA_ekg;
output[11:0] count;
output[7:0] data;

reg[2:0] state, nextstate;
reg cs_bar_DA_ekg, cs_bar_DA_ekg_int, LE_DAC, LE_DAC_int;
reg[11:0] count, count_int;
wire[7:0] data;

assign data = LE_DAC ? ekg_q : 8'hz;

parameter IDLE = 0;
parameter DAC0 = 1;
parameter DAC1 = 2;
parameter DAC2 = 3;
parameter DAC3 = 4;
parameter WAIT = 5;

always @ (posedge clk)
begin
    count <= count_int;
    cs_bar_DA_ekg <= cs_bar_DA_ekg_int;
    LE_DAC <= LE_DAC_int;

    if (!reset_sync) begin state <= IDLE; count_int <= 0; end
    else state <= nextstate;

    if (state == IDLE) count_int <= 0;
    else if (state == DAC3 && count == 4095) count_int <= 0;
    else if (state == DAC3) count_int <= count_int + 1;
end

always @ (state or sample_ekg)
begin
    cs_bar_DA_ekg_int = 1;
    LE_DAC_int = 0;

    case (state)
      IDLE:    begin
                   nextstate = WAIT;
                 end

        WAIT:    begin
                     if (sample_ekg) nextstate = DAC0;
                     else nextstate = WAIT;
                 end

        DAC0:    begin
                     LE_DAC_int = 1;
                     nextstate = DAC1;
                 end

        DAC1:    begin
                     LE_DAC_int = 1;
                     cs_bar_DA_ekg_int = 0;
                     nextstate = DAC2;
                 end

        DAC2:    begin
                     LE_DAC_int = 1;
                     nextstate = DAC3;
                 end

        DAC3:    begin
                     LE_DAC_int = 0;
                     nextstate = WAIT;
                 end

        default: nextstate = IDLE;
    endcase
end

endmodule
```

```
/*******************************************************************************
*     This file is owned and controlled by Xilinx and must be used             *
*     solely for design, simulation, implementation and creation of            *
*     design files limited to Xilinx devices or technologies. Use              *
*     with non-Xilinx devices or technologies is expressly prohibited          *
*     and immediately terminates your license.                                 *
*                                                                              *
*     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"            *
*     SOLELY FOR USE IN DEVELOPING PROGRAMS AND SOLUTIONS FOR                  *
*     XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE, OR INFORMATION          *
*     AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE, APPLICATION              *
*     OR STANDARD, XILINX IS MAKING NO REPRESENTATION THAT THIS                *
*     IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,                  *
*     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE         *
*     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY                 *
*     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE                  *
*     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR           *
*     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF          *
*     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS          *
*     FOR A PARTICULAR PURPOSE.                                                *
*                                                                              *
*     Xilinx products are not intended for use in life support                 *
*     appliances, devices, or systems. Use in such applications are            *
*     expressly prohibited.                                                    *
*                                                                              *
*     (c) Copyright 1995-2004 Xilinx, Inc.                                     *
*     All rights reserved.                                                     *
*******************************************************************************/
// The synopsys directives "translate_off/translate_on" specified below are
// supported by XST, FPGA Compiler II, Mentor Graphics and Synplicity synthesis
// tools. Ensure they are correct for your synthesis tool(s).

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

module ekg_simulator (
   A,
   SPO);    // synthesis black_box

input [11 : 0] A;
output [7 : 0] SPO;

// synopsys translate_off

   C_DIST_MEM_V7_1 #(
      12,   // c_addr_width
      "0",  // c_default_data
      1, // c_default_data_radix
      4096, // c_depth
      1, // c_enable_rlocs
      1, // c_generate_mif
      0, // c_has_clk
      0, // c_has_d
      0, // c_has_dpo
      0, // c_has_dpra
      0, // c_has_i_ce
      0, // c_has_qdpo
      0, // c_has_qdpo_ce
      0, // c_has_qdpo_clk
      0, // c_has_qdpo_rst
      0, // c_has_qdpo_srst
      0, // c_has_qspo
      0, // c_has_qspo_ce
      0, // c_has_qspo_rst
      0, // c_has_qspo_srst
      0, // c_has_rd_en
      1, // c_has_spo
      0, // c_has_spra
      0, // c_has_we
      0, // c_latency
      "ekg_simulator.mif", // c_mem_init_file
      0, // c_mem_type
      0, // c_mux_type
      0, // c_qce_joined
      0, // c_qualify_we
      1, // c_read_mif
      0, // c_reg_a_d_inputs
      0, // c_reg_dpra_input
      0, // c_sync_enable
      8) // c_width
   inst (
      .A(A),
      .SPO(SPO),
      .D(),
      .DPRA(),
      .SPRA(),
      .I_CE(),
      .QSPO_CE(),
      .WE(),
      .CLK(),
      .QDPO_CE(),
      .QDPO_CLK(),
      .RD_EN(),
      .QSPO_RST(),
      .QDPO_RST(),
      .QSPO_SRST(),
      .QDPO_SRST(),
      .QSPO(),
      .DPO(),
      .QDPO());


// synopsys translate_on

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

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

endmodule
```

```
module ekg_synchronizer(clk, reset, reset_sync);

input clk, reset;
output reset_sync;
reg x1, reset_sync;

always @ (posedge clk)
begin
    x1 <= reset;
    reset_sync <= x1;
end

endmodule
```

```verilog
module enable12andahalfHz(clk, reset_sync, sample_wireless);

input clk, reset_sync;
output sample_wireless;
reg sample_wireless;
reg[19:0] counter;

always @ (posedge clk or negedge reset_sync)
begin
   if(reset_sync == 0)
   begin
      sample_wireless <= 1'b0;
      counter <= 20'd0;
   end
   else if(counter == 20'd799999)      // 10 MHz --> 12.5 Hz
   begin
      sample_wireless <= 1'b1;
      counter <= 20'd0;
   end
   else
   begin
      sample_wireless <= 1'b0;
      counter <= counter + 1;
   end
end

endmodule
```

```
module enable250Hz(clk, reset_sync, sample);

input clk, reset_sync;
output sample;
reg sample;
reg[15:0] counter;

always @ (posedge clk or negedge reset_sync)
begin
   if(reset_sync == 0)
   begin
      sample <= 1'b0;
      counter <= 16'd0;
   end
   // else if(counter == 9'd4)
   else if(counter == 16'd40000)      // 10 MHz --> 10 kHz
   begin
      sample <= 1'b1;
      counter <= 16'd0;
   end
   else
   begin
      sample <= 1'b0;
      counter <= counter + 1;
   end
end

endmodule
```

```verilog
module enable360Hz(clk, reset_sync, sample_ekg);

input clk, reset_sync;
output sample_ekg;
reg sample_ekg;
reg[14:0] counter;

always @ (posedge clk or negedge reset_sync)
begin
   if(reset_sync == 0)
   begin
      sample_ekg <= 1'b0;
      counter <= 15'd0;
   end
   else if(counter == 15'd27778) //d111115)      // 10 MHz --> 360 Hz
   begin
      sample_ekg <= 1'b1;
      counter <= 15'd0;
   end
   else
   begin
      sample_ekg <= 1'b0;
      counter <= counter + 1;
   end
end

endmodule
```

```verilog
module filter(clk, reset_sync, filter_start, sram_q, rom1_q, rom2_q, filter_busy, ADC_start, sram_we, sram_addr,
 rom1_addr, rom2_addr, int_data);

input clk, reset_sync, filter_start;
input[7:0] sram_q, rom1_q, rom2_q;
output filter_busy, sram_we, ADC_start;
output[5:0] sram_addr, rom1_addr, rom2_addr;
output[7:0] int_data;

reg sram_we, sram_we_int, filter_busy, filter_busy_int, ADC_start, ADC_start_int;
reg LD_accum;
reg[5:0] sram_addr, sram_addr_recent, rom1_addr, rom2_addr, count, count_int;
reg[2:0] state, nextstate;
reg[15:0] Q1, Q2;
reg[7:0] int_data;
wire[15:0] product1, product2, D1, D2, Q;

parameter IDLE = 0;
parameter STORE0 = 1;
parameter STORE1 = 2;
parameter STORE2 = 3;
parameter STARTADC = 4;
parameter CONVOLVE0 = 5;
parameter CONVOLVE1 = 6;

always @ (posedge clk)
begin
    sram_we <= sram_we_int;
    filter_busy <= filter_busy_int;
    ADC_start <= ADC_start_int;
    rom1_addr <= count;
    rom2_addr <= count;
    sram_addr <= sram_addr_recent + count;

    if (!reset_sync) begin
        state <= IDLE;
        sram_addr_recent <= 0;
        count <= 0;
        end
    else state <= nextstate;

      if (LD_accum) begin Q1 <= D1; Q2 <= D2; end
       else begin Q1 <= Q1; Q2 <= Q2; end

    if (state == IDLE) begin
        count <= 6'b0;
        Q1 <= 16'b0;
        Q2 <= 16'b0;
        end
    else if (state == CONVOLVE1 && count == 63) begin
        count <= 6'b0;
        sram_addr_recent <= sram_addr_recent - 1;
        end
    else if (state == CONVOLVE1)
        count <= count + 1;
end

always @ (state or reset_sync or filter_start)
begin
    sram_we_int = 0;
    filter_busy_int = 1;
    ADC_start_int = 0;
    LD_accum = 0;

    case (state)
        IDLE:      begin
                       filter_busy_int = 0;
                       if (filter_start) nextstate = STORE0;
                       else nextstate = IDLE;
                   end

        STORE0:    begin
                       nextstate = STORE1;
                   end

        STORE1:    begin
                       sram_we_int = 1;
                       nextstate = STORE2;
                   end

        STORE2:    begin
                       nextstate = STARTADC;
                   end

        STARTADC:  begin
                       ADC_start_int = 1;
                       nextstate = CONVOLVE0;
                   end

        CONVOLVE0: begin
                       LD_accum = 1;
                       nextstate = CONVOLVE1;
                   end

        CONVOLVE1: begin
                       if (count == 6'b111111) begin
                 if (!Q[15] && (Q[14] || Q[13] || Q[12] || Q[11])) int_data = 8'b11111111;
                 else if (Q[15] && (!Q[14] || !Q[13] || !Q[12] || !Q[11])) int_data = 8'b00000000;
                 else int_data = {~Q[15], Q[10:4]};
                           nextstate = IDLE;
                           end
                       else nextstate = CONVOLVE0;
                   end

        default: nextstate = IDLE;
    endcase
end

multiplier multiplier1(sram_q, rom1_q, product1);
assign D1 = product1 + Q1;
multiplier multiplier2(sram_q, rom2_q, product2);
assign D2 = product2 + Q2;
multiplier_2cpmt multiplier_2cpmt1(Q1[15:8], Q2[15:8], Q);

endmodule
```

```
///////////////////////////////////////////////////////////////////////////////
//
// 6.111 FPGA Labkit -- Flash ROM Interface
//
// For Labkit Revision 004
//
//
// Created: January 22, 2005
// Author: Nathan Ickes
//
///////////////////////////////////////////////////////////////////////////////

`define FLASHOP_IDLE  2'b00
`define FLASHOP_READ  2'b01
`define FLASHOP_WRITE 2'b10

module flash_int(reset, clock, op, address, wdata, rdata, busy, flash_data,
       flash_address, flash_ce_b, flash_oe_b, flash_we_b,
       flash_reset_b, flash_sts, flash_byte_b);

   parameter access_cycles = 5;
   parameter reset_assert_cycles = 1000;
   parameter reset_recovery_cycles = 30;

   input reset, clock; // Reset and clock for the flash interface
   input [1:0] op; // Flash operation select (read, write, idle)
   input [22:0] address;
   input [15:0] wdata;
   output [15:0] rdata;
   output busy;
   inout [15:0] flash_data;
   output [23:0] flash_address;
   output flash_ce_b, flash_oe_b, flash_we_b;
   output flash_reset_b, flash_byte_b;
   input  flash_sts;

   reg [1:0] lop;
   reg [15:0] rdata;
   reg busy;
   reg [15:0] flash_wdata;
   reg flash_ddata;
   reg [23:0] flash_address;
   reg flash_oe_b, flash_we_b, flash_reset_b;

   assign flash_ce_b = flash_oe_b && flash_we_b;
   assign flash_byte_b = 1; // 1 = 16-bit mode (A0 ignored)

   assign flash_data = flash_ddata ? flash_wdata : 16'hZ;

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

   initial
     flash_reset_b <= 1'b1;

   reg [9:0] state;

   always @(posedge clock)
     if (reset)
       begin
     state <= 0;
     flash_reset_b <= 0;
     flash_we_b <= 1;
     flash_oe_b <= 1;
     flash_ddata <= 0;
     busy <= 1;
       end
     else if (flash_reset_b == 0)
       if (state == reset_assert_cycles)
    begin
       flash_reset_b <= 1;
       state <= 1023-reset_recovery_cycles;
    end
       else
    state <= state+1;
     else if ((state == 0) && !busy)
```

```
       // The flash chip and this state machine are both idle. Latch the user's
       // address and write data inputs. Deassert OE and WE, and stop driving
       // the data buss ourselves. If a flash operation (read or write) is
       // requested, move to the next state.
       begin
     flash_address <= {address, 1'b0};
     flash_we_b <= 1;
     flash_oe_b <= 1;
     flash_ddata <= 0;
     flash_wdata <= wdata;
     lop <= op;
     if (op != `FLASHOP_IDLE)
       begin
          busy <= 1;
          state <= state+1;
       end
     else
       busy <= 0;
       end
     else if ((state==0) && flash_sts)
       busy <= 0;
     else if (state == 1)
       // The first stage of a flash operation. The address bus is already set,
       // so, if this is a read, we assert OE. For a write, we start driving
       // the user's data onto the flash databus (the value was latched in the
       // previous state.
       begin
     if (lop == `FLASHOP_WRITE)
       flash_ddata <= 1;
     else if (lop == `FLASHOP_READ)
       flash_oe_b <= 0;
     state <= state+1;
       end
     else if (state == 2)
       // The second stage of a flash operation. Nothing to do for a read. For
       // a write, we assert WE.
       begin
     if (lop == `FLASHOP_WRITE)
       flash_we_b <= 0;
     state <= state+1;
       end
     else if (state == access_cycles+1)
       // The third stage of a flash operation. For a read, we latch the data
       // from the flash chip. For a write, we deassert WE.
       begin
     if (lop == `FLASHOP_WRITE)
       flash_we_b <= 1;
     if (lop == `FLASHOP_READ)
       rdata <= flash_data;
     state <= 0;
       end
     else
       begin
     if (!flash_sts)
       busy <= 1;
     state <= state+1;
       end

endmodule

```

```
///////////////////////////////////////////////////////////////////////////////
//
// 6.111 FPGA Labkit -- Flash Test Program
//
// For Labkit Revision 004
//
//
// Created: January 23, 2005, during a blizzard
// Author: Nathan Ickes
//
/////////////////////////////////////////////////////////////////////////////////

`timescale 1ns/1ns
//`include "flash_int.v"
//`include "test_fsm.v"
//`include "display.v"

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;

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

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

   ////////////////////////////////////////////////////////////////////////////
   //
   // Flash ROM Tester
   //
   ////////////////////////////////////////////////////////////////////////////
```

```

    wire [1:0] fop;
    wire [22:0] faddress;
    wire [15:0] fwdata, frdata;
    wire fbusy;
    wire [639:0] dots;
    wire [7:0] FPGA_ROM_data;
    wire [12:0] FPGA_ROM_addr;

    flash_int flashint1 (reset, clock_27mhz, fop, faddress, fwdata, frdata,
          fbusy, flash_data, flash_address, flash_ce_b,
          flash_oe_b, flash_we_b, flash_reset_b, 1'b1,
          flash_byte_b);

    test_fsm fsm1 (reset, clock_27mhz, fop, faddress, fwdata, frdata, fbusy,
         dots, FPGA_ROM_data, FPGA_ROM_addr);

    display disp1 (reset, clock_27mhz, disp_blank, disp_clock, disp_rs,
         disp_ce_b, disp_reset_b, disp_data_out, dots);

    romsound romsound1 (FPGA_ROM_addr, FPGA_ROM_data);

    assign analyzer1_clock = clock_27mhz;

    assign analyzer1_data = {7'b0000000,    // 15-9
              fbusy,         // 8
              flash_ce_b,    // 7
              flash_oe_b,    // 6
              flash_we_b,    // 5
              flash_reset_b, // 4
              flash_sts,     // 3
              fop,           // 2-1
              reset};        // 0

    assign analyzer2_data = flash_data;

    assign user1 = {flash_data,    // 31-16
          6'h00,         // 15-10
          fbusy,         // 9
          flash_ce_b,    // 8
          flash_oe_b,    // 7
          flash_we_b,    // 6
          flash_reset_b, // 5
          flash_sts,     // 4
          fop,           // 3-2
          reset,         // 1
          clock_27mhz};  // 0

    assign analyzer3_data = {9'b000000000,
              flash_address[22:16]};

    assign analyzer4_data = flash_address[15:0];

    assign user2 = {disp_blank,     // 31
          disp_clock,     // 30
          disp_rs,        // 29
          disp_ce_b,      // 28
          disp_reset_b,   // 27
          disp_data_out,  // 26
          3'b000,         // 27-23
          flash_address}; // 22-0

    //assign led = {6'b111111, ~fop};
    assign led = { dots[7:0]};

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

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

    // 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;

   // 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;

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

   // User I/Os
   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;

endmodule
```

```
module heartbeat_detector(clk, reset_sync, int_data, heartbeat);

input clk, reset_sync;
input[7:0] int_data;
output heartbeat;
reg heartbeat;

always @ (posedge clk)
begin
   if (!reset_sync) heartbeat <= 0;
   else if ((int_data[7] && int_data[6]) || (int_data[7] && !int_data[6] && int_data[5])) heartbeat <= 1;
   else heartbeat <= 0;
end

endmodule
```

```verilog
module heartrate_detector(clk, reset_sync, heartbeat, heartrate, convert);

input clk, reset_sync, heartbeat;
output[8:0] heartrate;
output convert;

wire[2:0] six;
wire[11:0] heartrate1;
reg convert;
reg[8:0] heartrate_int, heartrate;
reg[26:0] count10sec;
reg reset_count;

assign six = 3'b110;
assign heartrate1 = heartrate_int * six;

always @ (posedge clk)
begin
   if (!reset_sync) begin
      count10sec <= 0;
      reset_count <= 1;
      heartrate <= 9'b111111111;
      convert <= 0;
      end
   else if (count10sec == 27'd99999999) begin      // 10 MHz -> 0.1 Hz
      count10sec <= 0;
      heartrate <= heartrate1[8:0];
      reset_count <= 1;
      convert <= 1;
      end
   else begin
      count10sec <= count10sec + 1;
      reset_count <= 0;
      convert <= 0;
      end
end

always @ (posedge heartbeat or posedge reset_count)
begin
   if (reset_count) heartrate_int <= 0;
   else heartrate_int <= heartrate_int + 1;
end

endmodule
```

```
///////////////////////////////////////////////////////////////////////////////
//
// 6.111 FPGA Labkit -- Template Toplevel Module
//
// For Labkit Revision 004
//
//
// Created: October 31, 2004, from revision 003 file
// Author: Nathan Ickes
//
///////////////////////////////////////////////////////////////////////////////
//
// CHANGES FOR BOARD REVISION 004
//
// 1) Added signals for logic analyzer pods 2-4.
// 2) Expanded "tv_in_ycrcb" to 20 bits.
// 3) Renamed "tv_out_data" to "tv_out_i2c_data" and "tv_out_sclk" to
//    "tv_out_i2c_clock".
// 4) Reversed disp_data_in and disp_data_out signals, so that "out" is an
//    output of the FPGA, and "in" is an input.
//
// CHANGES FOR BOARD REVISION 003
//
// 1) Combined flash chip enables into a single signal, flash_ce_b.
//
// CHANGES FOR BOARD REVISION 002
//
// 1) Added SRAM clock feedback path input and output
// 2) Renamed "mousedata" to "mouse_data"
// 3) Renamed some ZBT memory signals. Parity bits are now incorporated into
//    the data bus, and the byte write enables have been combined into the
//    4-bit ram#_bwe_b bus.
// 4) Removed the "systemace_clock" net, since the SystemACE clock is now
//    hardwired on the PCB to the oscillator.
//
///////////////////////////////////////////////////////////////////////////////
//
// Complete change history (including bug fixes)
//
// 2005-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
   //
   ////////////////////////////////////////////////////////////////////////////

   // 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

   // 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

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

   wire reset_pos, reset_wireless;
   SRL16 reset_sr111 (.D(1'b0), .CLK(clock_27mhz), .Q(reset_pos),
         .A0(1'b1), .A1(1'b1), .A2(1'b1), .A3(1'b1));
   defparam reset_sr.INIT = 16'hFFFF;
   assign reset_wireless = ~reset_pos;


   ////////////////////////////////////////////////////////////////////////////
   //
   // RS-232 Interface Tester
   //
   ////////////////////////////////////////////////////////////////////////////

   assign rs232_txd = 1'b1;
   assign rs232_rts = 1'b1;

   wire [7:0] rxdata;
   wire [5:0] rx_state;
   wire clk_173khz;

   Clock_173kHz clock_173khz1(clock_27mhz, reset_wireless, clk_173khz);
   rs232_rx rs232_rx1(reset_wireless, clk_173khz, switch[5], rs232_rxd, rxdata, rx_state);
   // rs232_rxd and rs232_cts are inputs


   ////////////////////////////////////////////////////////////////////////////
   //
   // EKG System
   //
   ////////////////////////////////////////////////////////////////////////////

   wire reset, status, reset_ekg;
   wire cs_bar, r_wbar, cs_bar_DA, cs_bar_DA_ekg, heartbeat, abnormal;
   wire[3:0] digit2, digit1, digit0;
   wire[8:0] heartrate;

   assign status = user1[0];
   assign reset = switch[1];
   assign reset_ekg = switch[0];

   top top1(clock1, reset, status, cs_bar, r_wbar, cs_bar_DA, user1[31:24], heartbeat, heartrate,
      digit2, digit1, digit0, abnormal);
   ekg ekg1(clock1, reset_ekg, cs_bar_DA_ekg, user4[8:1], rxdata, user4[28:21], user4[20]);

   assign user1[1] = cs_bar;
   assign user1[2] = r_wbar;
   assign user1[3] = cs_bar_DA;
   assign user4[0] = cs_bar_DA_ekg;
   assign user4[31] = heartbeat;


   ///////////////////////////////////////////////////////////////////////////
   //
   // Logic Analyzer
   //
   ///////////////////////////////////////////////////////////////////////////

   // Logic Analyzer
   assign analyzer1_data = {1'b0, rx_state, rs232_rxd, rxdata};  //16 bits
   assign analyzer1_clock = clk_173khz;


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

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

   ///////////////////////////////////////////////////////////////////////////
   //
   // Flash ROM
   //
   ///////////////////////////////////////////////////////////////////////////

      wire [22:0] faddress;
      wire [15:0] fwdata, frdata;
   wire fbusy;

   flash_int flashint1 (reset_audio, clock_27mhz, 2'b01, faddress, fwdata, frdata,
         fbusy, flash_data, flash_address, flash_ce_b,
         flash_oe_b, flash_we_b, flash_reset_b, 1'b1,
         flash_byte_b);

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

      wire[3:0] vol;
      wire[39:0] vdisp;
      wire volume_up, volume_down, heart_beep;
   //assign reset_audio = ~(switch[2] || switch[3]);
   assign heart_beep = switch[2] ? (switch[3] ? (abnormal || heartbeat) : abnormal)
                     : (switch[3] ? heartbeat : 1'b0);
      audio audio1 (reset_audio, clock_27mhz, audio_reset_b, ac97_sdata_out,
       ac97_sdata_in, ac97_synch, ac97_bit_clock, {vol, 1'b0}, switch[4], frdata, faddress, heartrate);
      beeper beep1 (reset_audio, clock_27mhz, beep, heart_beep);
      debounce vol_up (reset_audio, clock_27mhz, button_up, volume_up);
      debounce vol_down (reset_audio, clock_27mhz, button_down, volume_down);
      volume vol1 (reset_audio, clock_27mhz, volume_up, volume_down, vol, vdisp);

   assign led[7:0] = {~abnormal, ~abnormal, ~abnormal, ~abnormal, ~abnormal, ~abnormal, ~abnormal, ~abnormal};


   ///////////////////////////////////////////////////////////////////////////
   //
   // Display
   //
   ////////////////////////////////////////////////////////////////////////////

      wire[639:0] dots, dots_heart, dots_vol;
   reg[79:0] volume_dots;
   wire[39:0] one, two, three, four, five, six, seven, eight, nine, zero, dash, blank, qmark;
   reg[39:0] digit2_display, digit1_display, digit0_display;

   assign zero  = 40'b00111110_01000001_01000001_01000001_00111110;
   assign one   = 40'b00000000_01000010_01111111_01000000_00000000;
   assign two   = 40'b01100010_01010001_01001001_01001001_01000110;
   assign three = 40'b00100010_01000001_01001001_01001001_00110110;
   assign four  = 40'b00011000_00010100_00010010_01111111_00010000;
   assign five  = 40'b00100111_01000101_01000101_01000101_00111001;
   assign six   = 40'b00111100_01001010_01001001_01001001_00110000;
   assign seven = 40'b00000001_01110001_00001001_00000101_00000011;
   assign eight = 40'b00110110_01001001_01001001_01001001_00110110;
   assign nine  = 40'b00000110_01001001_01001001_00101001_00011110;
   assign dash  = 40'b00001000_00001000_00001000_00001000_00001000;
   assign blank = 40'b00000000_00000000_00000000_00000000_00000000;
   assign qmark = 40'b00000010_00000001_01010001_00001001_00000110; //'?'

   assign dots = switch[7] ? dots_heart : dots_vol;

   always @ (digit2 or digit1 or digit0)
   begin
      case (digit2)
         4'b0000: digit2_display <= blank;      4'b0001: digit2_display <= one;
         4'b0010: digit2_display <= two;     4'b0011: digit2_display <= three;
         4'b0100: digit2_display <= four;    4'b0101: digit2_display <= five;
         4'b0110: digit2_display <= six;     4'b0111: digit2_display <= seven;
         4'b1000: digit2_display <= eight;      4'b1001: digit2_display <= nine;
         default:    digit2_display <= dash;
      endcase

      case (digit1)
         4'b0000: digit1_display <= zero;    4'b0001: digit1_display <= one;
         4'b0010: digit1_display <= two;     4'b0011: digit1_display <= three;
         4'b0100: digit1_display <= four;    4'b0101: digit1_display <= five;
         4'b0110: digit1_display <= six;     4'b0111: digit1_display <= seven;
         4'b1000: digit1_display <= eight;      4'b1001: digit1_display <= nine;
         default:    digit1_display <= dash;
      endcase

      case (digit0)
         4'b0000: digit0_display <= zero;    4'b0001: digit0_display <= one;
         4'b0010: digit0_display <= two;     4'b0011: digit0_display <= three;
         4'b0100: digit0_display <= four;    4'b0101: digit0_display <= five;
         4'b0110: digit0_display <= six;     4'b0111: digit0_display <= seven;
         4'b1000: digit0_display <= eight;      4'b1001: digit0_display <= nine;
         default:    digit0_display <= dash;
      endcase
   end

      always @(vol)
     case (vol)
         5'd15:   volume_dots <= {one, five};
         5'd14:   volume_dots <= {one, four};
         5'd13:   volume_dots <= {one, three};
         5'd12:   volume_dots <= {one, two};
         5'd11:   volume_dots <= {one, one};
         5'd10:   volume_dots <= {one, zero};
         5'd09:   volume_dots <= {blank, nine};
      5'd08:   volume_dots <= {blank, eight};
      5'd07:   volume_dots <= {blank, seven};
      5'd06:   volume_dots <= {blank, six};
      5'd05:   volume_dots <= {blank, five};
      5'd04:   volume_dots <= {blank, four};
      5'd03:   volume_dots <= {blank, three};
      5'd02:   volume_dots <= {blank, two};
      5'd01:   volume_dots <= {blank, one};
      5'd00:   volume_dots <= {blank, zero};
      default:    volume_dots <= {blank, qmark};
     endcase

      assign dots_heart = { 40'b01111111_00001000_00001000_00001000_01111111, // 'H'
         40'b01111111_01001001_01001001_01001001_01000001, // 'E'
         40'b01111100_00010010_00010001_00010010_01111100, // 'A'
         40'b01111111_00001001_00011001_00101001_01000110, // 'R'
         40'b00000001_00000001_01111111_00000001_00000001, // 'T'
         blank,
         40'b01111111_00001001_00011001_00101001_01000110, // 'R'
         40'b01111100_00010010_00010001_00010010_01111100, // 'A'
         40'b00000001_00000001_01111111_00000001_00000001, // 'T'
         40'b01111111_01001001_01001001_01001001_01000001, // 'E'
         40'b00000000_00110110_00110110_00000000_00000000, // ':'
         blank,
         digit2_display,
         digit1_display,
         digit0_display,
         blank
         };

   assign dots_vol = { 40'b00000111_00011000_01100000_00011000_00000111, // 'V'
         40'b00111110_01000001_01000001_01000001_00111110, // 'O'
         40'b01111111_01000000_01000000_01000000_01000000, // 'L'
         40'b00111111_01000000_01000000_01000000_00111111, // 'U'
         40'b01111111_00000010_00001100_00000010_01111111, // 'M'
         40'b01111111_01001001_01001001_01001001_01000001, // 'E'
         40'b00000000_00110110_00110110_00000000_00000000, // ':'
         40'b00000000_00000000_00000000_00000000_00000000, // ' '
         volume_dots,
         blank, blank, blank, blank, blank, blank
         };

      display disp(1'b0, clock_27mhz, disp_blank, disp_clock, disp_rs,
       disp_ce_b, disp_reset_b, disp_data_out, dots);

endmodule
```

```
module majorFSM(clk, reset_sync, sample, filter_busy, analog_busy, filter_start, DAC_start);

input clk, reset_sync, sample, filter_busy, analog_busy;
output filter_start, DAC_start;
reg DAC_start, filter_start;
reg[2:0] state, nextstate;

parameter IDLE = 0;
parameter WAIT = 1;
parameter DO_DAC0 = 2;
parameter DO_DAC1 = 3;
parameter DO_DAC2 = 4;
parameter DO_MEM0 = 5;
parameter DO_MEM1 = 6;
parameter DO_MEM2 = 7;

always @ (posedge clk)
begin
    if (!reset_sync) state <= IDLE;
    else state <= nextstate;
end

always @ (state or sample or analog_busy or filter_busy)
begin
    DAC_start = 0;
    filter_start = 0;

    case (state)
        IDLE:    begin
                     nextstate = WAIT;
                 end

        WAIT:    begin
                     if (sample) nextstate = DO_DAC0;
                     else nextstate = WAIT;
                 end

        DO_DAC0: begin
                     if (!analog_busy) nextstate = DO_DAC1;
                     else nextstate = DO_DAC0;
                 end

        DO_DAC1: begin
                     DAC_start = 1;
                     if (analog_busy) nextstate = DO_DAC2;
                     else nextstate = DO_DAC1;
                 end

        DO_DAC2: begin
                     if (!analog_busy) nextstate = DO_MEM0;
                     else nextstate = DO_DAC2;
                 end

        DO_MEM0: begin
                     if (!filter_busy) nextstate = DO_MEM1;
                     else nextstate = DO_MEM0;
                 end

        DO_MEM1: begin
                     filter_start = 1;
                     if (filter_busy) nextstate = DO_MEM2;
                     else nextstate = DO_MEM1;
                 end

        DO_MEM2: begin
                     if (!filter_busy && !analog_busy) nextstate = WAIT;
                     else nextstate = DO_MEM2;
                 end

        default: nextstate = IDLE;
    endcase
end

endmodule
```

```verilog
// 8bits*8bits Multiplier
// x is in twos complement format
// y is in sign magnitude format

module multiplier(x, y, z);

input[7:0] x, y;
// 16-bit output
// the biggest output magnitude is 2^14 (15 bits)
output[15:0] z;
wire[7:0] x_neg, y_neg;
wire[15:0] z_pos, z_neg, z;

// Use unsigned multiplication operation * in Verilog
// then account for the sign on the MSB
assign x_neg = (x == 8'b0) ? 8'b0 : (~x + 1);
assign y_neg = (y == 8'b0 || y == 8'b10000000) ? 8'b0 : ({~y[7], y[6:0]});
assign z_pos = (x[7] ? x_neg : x) * (y[7] ? y_neg : y);
assign z_neg = (z_pos == 8'b0) ? 8'b0 : (~z_pos + 1);
assign z = (x[7] ^ y[7]) ? z_neg : z_pos;

endmodule
```

```verilog
// 8bits*8bits Multiplier
// x,y,z are in twos complement format

module multiplier_2cpmt(x, y, z);

input[7:0] x, y;
// 16-bit output
// the biggest output magnitude is 2^14 (15 bits)
output[15:0] z;
wire[7:0] x_neg, y_neg;
wire[15:0] z_pos, z_neg, z;

// Use unsigned multiplication operation * in Verilog
// then account for the sign on the MSB
assign x_neg = (x == 8'b0) ? 8'b0 : (~x + 1);
assign y_neg = (y == 8'b0 || y == 8'b10000000) ? 8'b0 : (~y + 1);
assign z_pos = (x[7] ? x_neg : x) * (y[7] ? y_neg : y);
assign z_neg = (z_pos == 8'b0) ? 8'b0 : (~z_pos + 1);
assign z = (x[7] ^ y[7]) ? z_neg : z_pos;

endmodule
```

```
/*******************************************************************************
*     This file is owned and controlled by Xilinx and must be used             *
*     solely for design, simulation, implementation and creation of            *
*     design files limited to Xilinx devices or technologies. Use              *
*     with non-Xilinx devices or technologies is expressly prohibited          *
*     and immediately terminates your license.                                 *
*                                                                              *
*     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"            *
*     SOLELY FOR USE IN DEVELOPING PROGRAMS AND SOLUTIONS FOR                  *
*     XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE, OR INFORMATION          *
*     AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE, APPLICATION              *
*     OR STANDARD, XILINX IS MAKING NO REPRESENTATION THAT THIS                *
*     IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,                  *
*     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE         *
*     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY                 *
*     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE                  *
*     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR           *
*     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF          *
*     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS          *
*     FOR A PARTICULAR PURPOSE.                                                *
*                                                                              *
*     Xilinx products are not intended for use in life support                 *
*     appliances, devices, or systems. Use in such applications are            *
*     expressly prohibited.                                                    *
*                                                                              *
*     (c) Copyright 1995-2004 Xilinx, Inc.                                     *
*     All rights reserved.                                                     *
*******************************************************************************/
// The synopsys directives "translate_off/translate_on" specified below are
// supported by XST, FPGA Compiler II, Mentor Graphics and Synplicity synthesis
// tools. Ensure they are correct for your synthesis tool(s).

// You must compile the wrapper file rom1.v when simulating
// the core, rom1. When compiling the wrapper file, be sure to
// reference the XilinxCoreLib Verilog simulation library. For detailed
// instructions, please refer to the "CORE Generator Guide".

module rom1 (
   A,
   SPO);    // synthesis black_box

input [5 : 0] A;
output [7 : 0] SPO;

// synopsys translate_off

   C_DIST_MEM_V7_1 #(
      6, // c_addr_width
      "0",  // c_default_data
      1, // c_default_data_radix
      64,   // c_depth
      1, // c_enable_rlocs
      1, // c_generate_mif
      0, // c_has_clk
      0, // c_has_d
      0, // c_has_dpo
      0, // c_has_dpra
      0, // c_has_i_ce
      0, // c_has_qdpo
      0, // c_has_qdpo_ce
      0, // c_has_qdpo_clk
      0, // c_has_qdpo_rst
      0, // c_has_qdpo_srst
      0, // c_has_qspo
      0, // c_has_qspo_ce
      0, // c_has_qspo_rst
      0, // c_has_qspo_srst
      0, // c_has_rd_en
      1, // c_has_spo
      0, // c_has_spra
      0, // c_has_we
      0, // c_latency
      "rom1.mif", // c_mem_init_file
      0, // c_mem_type
      0, // c_mux_type
      0, // c_qce_joined
      0, // c_qualify_we
      1, // c_read_mif
      0, // c_reg_a_d_inputs
      0, // c_reg_dpra_input
      0, // c_sync_enable
      8) // c_width
   inst (
      .A(A),
      .SPO(SPO),
      .D(),
      .DPRA(),
      .SPRA(),
      .I_CE(),
      .QSPO_CE(),
      .WE(),
      .CLK(),
      .QDPO_CE(),
      .QDPO_CLK(),
      .RD_EN(),
      .QSPO_RST(),
      .QDPO_RST(),
      .QSPO_SRST(),
      .QDPO_SRST(),
      .QSPO(),
      .DPO(),
      .QDPO());


// synopsys translate_on

// FPGA Express black box declaration
// synopsys attribute fpga_dont_touch "true"
// synthesis attribute fpga_dont_touch of rom1 is "true"

// XST black box declaration
// box_type "black_box"
// synthesis attribute box_type of rom1 is "black_box"

endmodule


```

```
/*******************************************************************************
*     This file is owned and controlled by Xilinx and must be used             *
*     solely for design, simulation, implementation and creation of            *
*     design files limited to Xilinx devices or technologies. Use              *
*     with non-Xilinx devices or technologies is expressly prohibited          *
*     and immediately terminates your license.                                 *
*                                                                              *
*     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"            *
*     SOLELY FOR USE IN DEVELOPING PROGRAMS AND SOLUTIONS FOR                  *
*     XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE, OR INFORMATION          *
*     AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE, APPLICATION              *
*     OR STANDARD, XILINX IS MAKING NO REPRESENTATION THAT THIS                *
*     IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,                  *
*     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE         *
*     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY                 *
*     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE                  *
*     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR           *
*     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF          *
*     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS          *
*     FOR A PARTICULAR PURPOSE.                                                *
*                                                                              *
*     Xilinx products are not intended for use in life support                 *
*     appliances, devices, or systems. Use in such applications are            *
*     expressly prohibited.                                                    *
*                                                                              *
*     (c) Copyright 1995-2004 Xilinx, Inc.                                     *
*     All rights reserved.                                                     *
*******************************************************************************/
// The synopsys directives "translate_off/translate_on" specified below are
// supported by XST, FPGA Compiler II, Mentor Graphics and Synplicity synthesis
// tools. Ensure they are correct for your synthesis tool(s).

// You must compile the wrapper file rom2.v when simulating
// the core, rom2. When compiling the wrapper file, be sure to
// reference the XilinxCoreLib Verilog simulation library. For detailed
// instructions, please refer to the "CORE Generator Guide".

module rom2 (
   A,
   SPO);    // synthesis black_box

input [5 : 0] A;
output [7 : 0] SPO;

// synopsys translate_off

   C_DIST_MEM_V7_1 #(
      6, // c_addr_width
      "0",  // c_default_data
      1, // c_default_data_radix
      64,   // c_depth
      1, // c_enable_rlocs
      1, // c_generate_mif
      0, // c_has_clk
      0, // c_has_d
      0, // c_has_dpo
      0, // c_has_dpra
      0, // c_has_i_ce
      0, // c_has_qdpo
      0, // c_has_qdpo_ce
      0, // c_has_qdpo_clk
      0, // c_has_qdpo_rst
      0, // c_has_qdpo_srst
      0, // c_has_qspo
      0, // c_has_qspo_ce
      0, // c_has_qspo_rst
      0, // c_has_qspo_srst
      0, // c_has_rd_en
      1, // c_has_spo
      0, // c_has_spra
      0, // c_has_we
      0, // c_latency
      "rom2.mif", // c_mem_init_file
      0, // c_mem_type
      0, // c_mux_type
      0, // c_qce_joined
      0, // c_qualify_we
      1, // c_read_mif
      0, // c_reg_a_d_inputs
      0, // c_reg_dpra_input
      0, // c_sync_enable
      8) // c_width
   inst (
      .A(A),
      .SPO(SPO),
      .D(),
      .DPRA(),
      .SPRA(),
      .I_CE(),
      .QSPO_CE(),
      .WE(),
      .CLK(),
      .QDPO_CE(),
      .QDPO_CLK(),
      .RD_EN(),
      .QSPO_RST(),
      .QDPO_RST(),
      .QSPO_SRST(),
      .QDPO_SRST(),
      .QSPO(),
      .DPO(),
      .QDPO());


// synopsys translate_on

// FPGA Express black box declaration
// synopsys attribute fpga_dont_touch "true"
// synthesis attribute fpga_dont_touch of rom2 is "true"

// XST black box declaration
// box_type "black_box"
// synthesis attribute box_type of rom2 is "black_box"

endmodule
```

```
/*******************************************************************************
*     This file is owned and controlled by Xilinx and must be used             *
*     solely for design, simulation, implementation and creation of            *
*     design files limited to Xilinx devices or technologies. Use              *
*     with non-Xilinx devices or technologies is expressly prohibited          *
*     and immediately terminates your license.                                 *
*                                                                              *
*     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"            *
*     SOLELY FOR USE IN DEVELOPING PROGRAMS AND SOLUTIONS FOR                  *
*     XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE, OR INFORMATION          *
*     AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE, APPLICATION              *
*     OR STANDARD, XILINX IS MAKING NO REPRESENTATION THAT THIS                *
*     IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,                  *
*     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE         *
*     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY                 *
*     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE                  *
*     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR           *
*     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF          *
*     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS          *
*     FOR A PARTICULAR PURPOSE.                                                *
*                                                                              *
*     Xilinx products are not intended for use in life support                 *
*     appliances, devices, or systems. Use in such applications are            *
*     expressly prohibited.                                                    *
*                                                                              *
*     (c) Copyright 1995-2004 Xilinx, Inc.                                     *
*     All rights reserved.                                                     *
*******************************************************************************/
// The synopsys directives "translate_off/translate_on" specified below are
// supported by XST, FPGA Compiler II, Mentor Graphics and Synplicity synthesis
// tools. Ensure they are correct for your synthesis tool(s).

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

module romsound (
   A,
   SPO);    // synthesis black_box

input [12 : 0] A;
output [7 : 0] SPO;

// synopsys translate_off

   C_DIST_MEM_V7_1 #(
      13,   // c_addr_width
      "0",  // c_default_data
      1, // c_default_data_radix
      8192, // c_depth
      1, // c_enable_rlocs
      1, // c_generate_mif
      0, // c_has_clk
      0, // c_has_d
      0, // c_has_dpo
      0, // c_has_dpra
      0, // c_has_i_ce
      0, // c_has_qdpo
      0, // c_has_qdpo_ce
      0, // c_has_qdpo_clk
      0, // c_has_qdpo_rst
      0, // c_has_qdpo_srst
      0, // c_has_qspo
      0, // c_has_qspo_ce
      0, // c_has_qspo_rst
      0, // c_has_qspo_srst
      0, // c_has_rd_en
      1, // c_has_spo
      0, // c_has_spra
      0, // c_has_we
      0, // c_latency
      "romsound.mif",   // c_mem_init_file
      0, // c_mem_type
      0, // c_mux_type
      0, // c_qce_joined
      0, // c_qualify_we
      1, // c_read_mif
      0, // c_reg_a_d_inputs
      0, // c_reg_dpra_input
      0, // c_sync_enable
      8) // c_width
   inst (
      .A(A),
      .SPO(SPO),
      .D(),
      .DPRA(),
      .SPRA(),
      .I_CE(),
      .QSPO_CE(),
      .WE(),
      .CLK(),
      .QDPO_CE(),
      .QDPO_CLK(),
      .RD_EN(),
      .QSPO_RST(),
      .QDPO_RST(),
      .QSPO_SRST(),
      .QDPO_SRST(),
      .QSPO(),
      .DPO(),
      .QDPO());


// synopsys translate_on

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

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

endmodule
```

```
///////////////////////////////////////////////////////////////////////////////
//
// 6.111 FPGA Labkit -- RS232 State Machine
//
//
// Created: May 9, 2005
// Author: Amy Tang
//
///////////////////////////////////////////////////////////////////////////////
`define STATUS_RESET             4'h0
`define STATUS_READ_ID           4'h1
`define STATUS_CLEAR_LOCKS       4'h2



module rs232_rx (reset, clock, sw5, rs232_rxd, rxdata, state);

   input reset, clock;
   input rs232_rxd, sw5;
   output [7:0] rxdata;
   output [5:0] state;

   reg [7:0] rxdata;
   reg [2:0] ptr;       // serial bit ptr
   reg b0, b1, b2, b3, b4, b5, b6, b7;

   ////////////////////////////////////////////////////////////////////////////
   //
   // State Machine
   //
   ////////////////////////////////////////////////////////////////////////////

   // internal state
      reg [5:0] state;

   parameter IDLE = 0;
   parameter START1 = 1;
   parameter START2 = 2;
   parameter WRITE1 = 3;
   parameter WRITE2 = 4;
   parameter WRITE3 = 5;
   parameter STOP1 = 6;
   parameter STOP2 = 7;
   parameter STOP3 = 8;

   /*initial begin
      state <= IDLE;
      rxdata <= 0;         //rxd changes once all 8 bits are received
      ptr<= 0;
      b0 <= 0;    b1 <= 0; b2 <= 0;    b3 <= 0;
      b4 <= 0;    b5 <= 0; b6 <= 0;    b7 <= 0;
   end*/

   // Combination Block for always next block
   always @ (posedge clock or negedge reset) begin
         if (!reset) begin
         state <=IDLE;
         rxdata <= 0;
         ptr<= 0;
         b0 <= 0;    b1 <= 0; b2 <= 0;    b3 <= 0;
         b4 <= 0;    b5 <= 0; b6 <= 0;    b7 <= 0;
         end
         else if (sw5)
        case (state)
            IDLE:
               begin
               if (!rs232_rxd)   state <= START1;  // start bit detected; active low
               else state <= IDLE;
               end

            START1:
               begin
            state <= START2;
               end
         START2:
               begin
            state <= WRITE1;
               end

         WRITE1:
               begin
               state <= WRITE2;
               end

         WRITE2:
               begin
            if (ptr == 0)  b0 <= rs232_rxd;
            else if (ptr == 1)   b1 <= rs232_rxd;
            else if (ptr == 2)   b2 <= rs232_rxd;
            else if (ptr == 3)   b3 <= rs232_rxd;
            else if (ptr == 4)   b4 <= rs232_rxd;
            else if (ptr == 5)   b5 <= rs232_rxd;
            else if (ptr == 6)   b6 <= rs232_rxd;
            else if (ptr == 7)   b7 <= rs232_rxd;

               state <= WRITE3;
               end

         WRITE3:
               begin
            if (ptr == 3'd7) begin
               rxdata <= {b7, b6, b5, b4, b3, b2, b1, b0};
               ptr <= 0;
               state <= STOP1;
               end
            else begin
               ptr <= ptr +1;
               state <= WRITE1;
                  end
             end
         STOP1:
                  begin
                  state <= STOP2;
                  end

         STOP2:
                  begin
                  state <= STOP3;
                  end

         STOP3:
                  begin
                  state <= IDLE;
               end
         default: state <= IDLE;
      endcase // case(state)
      end // always @ (clock)
endmodule

```

```
/*******************************************************************************
*     This file is owned and controlled by Xilinx and must be used             *
*     solely for design, simulation, implementation and creation of            *
*     design files limited to Xilinx devices or technologies. Use              *
*     with non-Xilinx devices or technologies is expressly prohibited          *
*     and immediately terminates your license.                                 *
*                                                                              *
*     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"            *
*     SOLELY FOR USE IN DEVELOPING PROGRAMS AND SOLUTIONS FOR                  *
*     XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE, OR INFORMATION         *
*     AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE, APPLICATION              *
*     OR STANDARD, XILINX IS MAKING NO REPRESENTATION THAT THIS                *
*     IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,                  *
*     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE         *
*     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY                 *
*     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE                  *
*     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR           *
*     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF          *
*     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS          *
*     FOR A PARTICULAR PURPOSE.                                                *
*                                                                              *
*     Xilinx products are not intended for use in life support                 *
*     appliances, devices, or systems. Use in such applications are            *
*     expressly prohibited.                                                    *
*                                                                              *
*     (c) Copyright 1995-2004 Xilinx, Inc.                                     *
*     All rights reserved.                                                     *
*******************************************************************************/
// The synopsys directives "translate_off/translate_on" specified below are
// supported by XST, FPGA Compiler II, Mentor Graphics and Synplicity synthesis
// tools. Ensure they are correct for your synthesis tool(s).

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

module sram (
   A,
   CLK,
   D,
   WE,
   SPO);    // synthesis black_box

input [5 : 0] A;
input CLK;
input [7 : 0] D;
input WE;
output [7 : 0] SPO;

// synopsys translate_off

   C_DIST_MEM_V7_1 #(
      6, // c_addr_width
      "0",  // c_default_data
      1, // c_default_data_radix
      64,   // c_depth
      1, // c_enable_rlocs
      1, // c_generate_mif
      1, // c_has_clk
      1, // c_has_d
      0, // c_has_dpo
      0, // c_has_dpra
      0, // c_has_i_ce
      0, // c_has_qdpo
      0, // c_has_qdpo_ce
      0, // c_has_qdpo_clk
      0, // c_has_qdpo_rst
      0, // c_has_qdpo_srst
      0, // c_has_qspo
      0, // c_has_qspo_ce
      0, // c_has_qspo_rst
      0, // c_has_qspo_srst
      0, // c_has_rd_en
      1, // c_has_spo
      0, // c_has_spra
      1, // c_has_we
      0, // c_latency
      "sram.mif", // c_mem_init_file
      1, // c_mem_type
      0, // c_mux_type
      0, // c_qce_joined
      0, // c_qualify_we
      0, // c_read_mif
      0, // c_reg_a_d_inputs
      0, // c_reg_dpra_input
      0, // c_sync_enable
      8) // c_width
   inst (
      .A(A),
      .CLK(CLK),
      .D(D),
      .WE(WE),
      .SPO(SPO),
      .DPRA(),
      .SPRA(),
      .I_CE(),
      .QSPO_CE(),
      .QDPO_CE(),
      .QDPO_CLK(),
      .RD_EN(),
      .QSPO_RST(),
      .QDPO_RST(),
      .QSPO_SRST(),
      .QDPO_SRST(),
      .QSPO(),
      .DPO(),
      .QDPO());


// synopsys translate_on

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

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

endmodule
```

```verilog
module synchronizer(clk, reset, status, reset_sync, status_sync);

input clk, reset, status;
output reset_sync, status_sync;
reg x1, x2, reset_sync, status_sync;

always @ (posedge clk)
begin
    x1 <= reset;
    x2 <= status;
    reset_sync <= x1;
    status_sync <= x2;
end

endmodule
```

```
///////////////////////////////////////////////////////////////////
//
// 6.111 FPGA Labkit -- Flash Tester State Machine
//
//
// Created: January 23, 2005
// Author: Nathan Ickes
//
///////////////////////////////////////////////////////////////////

`define FLASHOP_IDLE  2'b00
`define FLASHOP_READ  2'b01
`define FLASHOP_WRITE 2'b10

`define STATUS_RESET             4'h0
`define STATUS_READ_ID           4'h1
`define STATUS_CLEAR_LOCKS       4'h2
`define STATUS_ERASING           4'h3
`define STATUS_WRITING           4'h4
`define STATUS_READING           4'h5
`define STATUS_SUCCESS           4'h6
`define STATUS_BAD_MANUFACTURER  4'h7
`define STATUS_BAD_SIZE          4'h8
`define STATUS_LOCK_BIT_ERROR    4'h9
`define STATUS_ERASE_BLOCK_ERROR 4'hA
`define STATUS_WRITE_ERROR       4'hB
`define STATUS_READ_WRONG_DATA   4'hC

//`define NUM_BLOCKS 128
`define NUM_BLOCKS 128
`define BLOCK_SIZE 64*1024
`define LAST_BLOCK_ADDRESS ((`NUM_BLOCKS-1)*`BLOCK_SIZE)
`define LAST_ADDRESS (`NUM_BLOCKS*`BLOCK_SIZE-1)

`define NUM_BLOCKS2 2
`define BLOCK_SIZE2 64*1024
`define LAST_BLOCK_ADDRESS2 ((`NUM_BLOCKS2-1)*`BLOCK_SIZE2)
`define LAST_ADDRESS2 (`NUM_BLOCKS2*`BLOCK_SIZE2-1)


`define INIT_FLASH_ADDRESS   23'd57351   // 23'h0, 23'd8192
`define LAST_FLASH_ADDRESS (`INIT_FLASH_ADDRESS+23'd8192)


module test_fsm (reset, clock, fop, faddress, fwdata, frdata, fbusy, dots, FPGA_ROM_data, FPGA_ROM_addr);

   input reset, clock;
   output [1:0] fop;
   output [22:0] faddress;
   output [15:0] fwdata;
   input [15:0]  frdata;
   input fbusy;
   output [639:0] dots;
   input [7:0] FPGA_ROM_data;
   output [12:0] FPGA_ROM_addr;

   reg [1:0] fop;
   reg [22:0] faddress;
   reg [15:0] fwdata;
   reg [639:0] dots;
   reg [12:0] FPGA_ROM_addr;

   ////////////////////////////////////////////////////////////////////////////
   //
   // State Machine
   //
   ////////////////////////////////////////////////////////////////////////////

   reg [7:0] state;
   reg [3:0] status;

   always @(posedge clock)
     if (reset)
       begin
     state <= 0;
     status <= `STATUS_RESET;
     faddress <= 0;
     FPGA_ROM_addr <= 0;
     fop <= `FLASHOP_IDLE;
```

```
       end
     else if (!fbusy && (fop == `FLASHOP_IDLE))
       case (state)
    8'h00:
      begin
         // Issue "read id codes" command
         status <= `STATUS_READ_ID;
         faddress <= 0;
         fwdata <= 16'h0090;
         fop <= `FLASHOP_WRITE;
         state <= state+1;
      end

    8'h01:
      begin
         // Read manufacturer code
         faddress <= 0;
         fop <= `FLASHOP_READ;
         state <= state+1;
      end

    8'h02:
      if (frdata != 16'h0089) // 16'h0089 = Intel
        status <= `STATUS_BAD_MANUFACTURER;
      else
        begin
      // Read the device size code
      faddress <= 1;
      fop <= `FLASHOP_READ;
      state <= state+1;
        end

    8'h03:
      if (frdata != 16'h0018) // 16'h0018 = 128Mbit
        status <= `STATUS_BAD_SIZE;
      else
        begin
      faddress <= 0;
      fwdata <= 16'hFF;
      fop <= `FLASHOP_WRITE;
      state <= state+1;
        end
    8'h04:
      begin
         // Issue "clear lock bits" command
         status <= `STATUS_CLEAR_LOCKS;
         faddress <= 0;
         fwdata <= 16'h60;
         fop <= `FLASHOP_WRITE;
         state <= state+1;
      end
    8'h05:
      begin
         // Issue "confirm clear lock bits" command
         faddress <= 0;
         fwdata <= 16'hD0;
         fop <= `FLASHOP_WRITE;
         state <= state+1;
      end
    8'h06:
      begin
         // Read status
         faddress <= 0;
         fop <= `FLASHOP_READ;
         state <= state+1;
      end
    8'h07:
      if (frdata[7] == 1) // Done clearing lock bits
        if (frdata[6:1] == 0) // No errors
          begin
        faddress <= `INIT_FLASH_ADDRESS;
        fop <= `FLASHOP_IDLE;
        state <= state+5;
          end
        else
          status <= `STATUS_LOCK_BIT_ERROR;
      else // Still busy, reread status register
        begin
      faddress <= 0;
      fop <= `FLASHOP_READ;
        end
/*
    ////////////////////////////////////////////////////////////////////
    // Block Erase Sequence
    ////////////////////////////////////////////////////////////////////

    8'h08:
      begin
         status <= `STATUS_ERASING;
         fwdata <= 16'h20; // Issue "erase block" command
         fop <= `FLASHOP_WRITE;
         state <= state+1;
      end
    8'h09:
      begin
         fwdata <= 16'hD0; // Issue "confirm erase" command
         fop <= `FLASHOP_WRITE;
         state <= state+1;
      end
    8'h0A:
      begin
         fop <= `FLASHOP_READ;
         state <= state+1;
      end
    8'h0B:
      if (frdata[7] == 1) // Done erasing block
        if (frdata[6:1] == 0) // No errors
          if (faddress != 23'h7F0000) // `LAST_BLOCK_ADDRESS)
        //if (faddress != `LAST_BLOCK_ADDRESS) // `LAST_BLOCK_ADDRESS)
        //if (faddress != 23'h7F0000) // `LAST_BLOCK_ADDRESS)
        //if (faddress != 23'h100) // `LAST_BLOCK_ADDRESS)
       begin
          faddress <= faddress+`BLOCK_SIZE;
          fop <= `FLASHOP_IDLE;
          state <= state-3;
       end
          else
       begin
          faddress <= `INIT_FLASH_ADDRESS;
          FPGA_ROM_addr <= 0;
          fop <= `FLASHOP_IDLE;
          state <= state+1;
       end
        else // Erase error detected
          status <= `STATUS_ERASE_BLOCK_ERROR;
      else // Still busy
        fop <= `FLASHOP_READ;

*/
    ////////////////////////////////////////////////////////////////////
    // Write Addresses to All Locations
    ////////////////////////////////////////////////////////////////////

    8'h0C:
      begin
         status <= `STATUS_WRITING;
         fwdata <= 16'h40; // Issue "setup write" command
         fop <= `FLASHOP_WRITE;
         state <= state+1;
      end

    8'h0D:
      begin
       //fwdata <= 16'b10101; // Finish write
       fwdata <= {8'b0, FPGA_ROM_data}; // Finish write
         fop <= `FLASHOP_WRITE;
         state <= state+1;
      end
    8'h0E:
      begin
         // Read status register
         fop <= `FLASHOP_READ;
         state <= state+1;
      end
    8'h0F:
      if (frdata[7] == 1) // Done writing
        if (frdata[6:1] == 0) // No errors
        if (faddress != `LAST_FLASH_ADDRESS) // `LAST_ADDRESS)
        //if (faddress != {20'b0, FPGA_ROM_addr}) // `LAST_ADDRESS)
        //if (faddress != 23'h7FFFFF) // `LAST_ADDRESS)  , if (faddress != 23'h20000)
       begin
          faddress <= faddress+1;
          FPGA_ROM_addr <= FPGA_ROM_addr+1;
          fop <= `FLASHOP_IDLE;
          state <= state-3;
       end
          else
       begin
          faddress <= `INIT_FLASH_ADDRESS;
          FPGA_ROM_addr <= 0;
          fop <= `FLASHOP_IDLE;
          state <= state+1;
       end
        else // Write error detected
          status <= `STATUS_WRITE_ERROR;
      else // Still busy
        fop <= `FLASHOP_READ;

    ////////////////////////////////////////////////////////////////////////
    // Read back data
    ////////////////////////////////////////////////////////////////////////

    8'h10:
      begin
         status <= `STATUS_READING;
         fwdata <= 16'hFF; // Issue "read array" command
         fop <= `FLASHOP_WRITE;
         faddress <= `INIT_FLASH_ADDRESS;
       FPGA_ROM_addr <= 0;
         state <= state+1;
      end
    8'h11:
      begin
         fop <= `FLASHOP_READ;
         state <= state+1;
      end
    8'h12:
      if (frdata == {8'b0, FPGA_ROM_data})
         if (faddress == `LAST_FLASH_ADDRESS)
      //if (faddress == `LAST_ADDRESS2)
      //if (faddress == 23'h7FFFFF)
          begin
        fop <= `FLASHOP_IDLE;
        faddress <= `INIT_FLASH_ADDRESS;
        FPGA_ROM_addr <= 0;
        state <= state+5;
          end
        else
          begin
        faddress <= faddress+1;
        FPGA_ROM_addr <= FPGA_ROM_addr+1;
        fop <= `FLASHOP_READ;
          end
      else
        status <= `STATUS_READ_WRONG_DATA;
  /*
    ////////////////////////////////////////////////////////////////////////
    // Erase the chip again
    ////////////////////////////////////////////////////////////////////////

    8'h13:
      begin
         status <= `STATUS_ERASING;
         fwdata <= 16'h20; // Issue "erase block" command
         fop <= `FLASHOP_WRITE;
         state <= state+1;
      end
    8'h14:
      begin
         fwdata <= 16'hD0; // Issue "confirm erase" command
         fop <= `FLASHOP_WRITE;
         state <= state+1;
      end
    8'h15:
      begin
         fop <= `FLASHOP_READ;
         state <= state+1;
      end
    8'h16:
      if (frdata[7] == 1) // Done erasing block
        if (frdata[6:1] == 0) // No errors
          if (faddress != 23'h20000) // `LAST_BLOCK_ADDRESS)
        //if (faddress != `LAST_BLOCK_ADDRESS) // `LAST_BLOCK_ADDRESS)
        // if (faddress != 23'h7F0000) // `LAST_BLOCK_ADDRESS)
        // if (faddress != 23'h100) // `LAST_BLOCK_ADDRESS)
       begin
          faddress <= faddress+`BLOCK_SIZE;
          fop <= `FLASHOP_IDLE;
          state <= state-3;
       end
          else
       begin
          faddress <= 0;
          fop <= `FLASHOP_IDLE;
          state <= state+1;
       end
        else // Erase error detected
          status <= `STATUS_ERASE_BLOCK_ERROR;
      else // Still busy
        fop <= `FLASHOP_READ;
*/
    ///////////////////////////////////////////////////////////////////////
    // End of test: declare success
    ///////////////////////////////////////////////////////////////////////

    8'h17:
      begin
         status <= `STATUS_SUCCESS;
         fop <= `FLASHOP_IDLE;
      end

       endcase
     else
       fop <= `FLASHOP_IDLE;

   ///////////////////////////////////////////////////////////////////////////
   //
   // Status display
   //
   ///////////////////////////////////////////////////////////////////////////
   //
   // "Reset     ------" --> During reset
   // "Read ID   000000" --> While reading ID codes
   // "Clr locks 000000" --> While clearing block locks
   // "Erase     000000" --> While erasing
   // "Write     000000" --> While writing
   // "Read      000000" --> While reading
   // " *** PASSED *** " --> If the entire test completes with no errors
   // "Err: Manuf  0000" --> If an incorrect manufacturer code is read
   // "Err: Size   0000" --> If an incorrect size code is read
   // "Err: Locks  0000" --> If an error is detected when clearing the block lock bits
   // "Err: Erase  0000"
   // "Err: 000000 0000"


   // Rd  000000  0000
   // Wr  000000  0000
   // Id  000000  0000
   //

   function [39:0] nib2char;
      input [3:0] nib;
      begin
    case (nib)
      4'h0: nib2char = 40'b00111110_01010001_01001001_01000101_00111110;
      4'h1: nib2char = 40'b00000000_01000010_01111111_01000000_00000000;
      4'h2: nib2char = 40'b01100010_01010001_01001001_01001001_01000110;
      4'h3: nib2char = 40'b00100010_01000001_01001001_01001001_00110110;
      4'h4: nib2char = 40'b00011000_00010100_00010010_01111111_00010000;
      4'h5: nib2char = 40'b00100111_01000101_01000101_01000101_00111001;
      4'h6: nib2char = 40'b00111100_01001010_01001001_01001001_00110000;
      4'h7: nib2char = 40'b00000001_01110001_00001001_00000101_00000011;
      4'h8: nib2char = 40'b00110110_01001001_01001001_01001001_00110110;
      4'h9: nib2char = 40'b00000110_01001001_01001001_00101001_00011110;
      4'hA: nib2char = 40'b01111110_00001001_00001001_00001001_01111110;
      4'hB: nib2char = 40'b01111111_01001001_01001001_01001001_00110110;
      4'hC: nib2char = 40'b00111110_01000001_01000001_01000001_00100010;
      4'hD: nib2char = 40'b01111111_01000001_01000001_01000001_00111110;
      4'hE: nib2char = 40'b01111111_01001001_01001001_01001001_01000001;
      4'hF: nib2char = 40'b01111111_00001001_00001001_00001001_00000001;
    endcase
      end
   endfunction

   wire [159:0] data_dots;
   assign data_dots = {nib2char(frdata[15:12]), nib2char(frdata[11:8]),
             nib2char(frdata[7:4]), nib2char(frdata[3:0])};

   wire [239:0] address_dots;
   assign address_dots = {nib2char({ 1'b0, faddress[22:20]}),
           nib2char(faddress[19:16]),
           nib2char(faddress[15:12]),
           nib2char(faddress[11:8]),
           nib2char(faddress[7:4]),
           nib2char(faddress[3:0])};

   always @(status or address_dots or data_dots)
     case (status)
       `STATUS_RESET:
    dots <= {40'b01111111_00001001_00011001_00101001_01000110, // R
        40'b01111111_01001001_01001001_01001001_01000001, // E
        40'b00100110_01001001_01001001_01001001_00110010, // S
        40'b01111111_01001001_01001001_01001001_01000001, // E
        40'b00000001_00000001_01111111_00000001_00000001, // T
        40'b00000000_00000000_00000000_00000000_00000000, //
        40'b00000000_00000000_00000000_00000000_00000000, //
        40'b00000000_00000000_00000000_00000000_00000000, //
        40'b00000000_00000000_00000000_00000000_00000000, //
        40'b00000000_00000000_00000000_00000000_00000000, //
        40'b00001000_00001000_00001000_00001000_00001000, // -
        40'b00001000_00001000_00001000_00001000_00001000, // -
        40'b00001000_00001000_00001000_00001000_00001000, // -
        40'b00001000_00001000_00001000_00001000_00001000, // -
        40'b00001000_00001000_00001000_00001000_00001000, // -
        40'b00001000_00001000_00001000_00001000_00001000};// -
       `STATUS_READ_ID:
    dots <= {40'b01111111_00001001_00011001_00101001_01000110, // R
        40'b01111111_01001001_01001001_01001001_01000001, // E
        40'b01111110_00001001_00001001_00001001_01111110, // A
        40'b01111111_01000001_01000001_01000001_00111110, // D
        40'b00000000_00000000_00000000_00000000_00000000, //
        40'b00000000_01000001_01111111_01000001_00000000, // I
        40'b01111111_01000001_01000001_01000001_00111110, // D
        40'b00000000_00000000_00000000_00000000_00000000, //
        40'b00000000_00000000_00000000_00000000_00000000, //
        40'b00000000_00000000_00000000_00000000_00000000, //
        address_dots};
       `STATUS_CLEAR_LOCKS:
    dots <= {40'b00111110_01000001_01000001_01000001_00100010, // C
        40'b01111111_01000000_01000000_01000000_01000000, // L
        40'b01111111_00001001_00011001_00101001_01000110, // R
        40'b00000000_00000000_00000000_00000000_00000000, //
        40'b01111111_01000000_01000000_01000000_01000000, // L
        40'b00111110_01000001_01000001_01000001_00111110, // O
        40'b00111110_01000001_01000001_01000001_00100010, // C
        40'b01111111_00001000_00010100_00100010_01000001, // K
        40'b00100110_01001001_01001001_01001001_00110010, // S
        40'b00000000_00000000_00000000_00000000_00000000, //
        address_dots};
       `STATUS_ERASING:
    dots <= {40'b01111111_01001001_01001001_01001001_01000001, // E
        40'b01111111_00001001_00011001_00101001_01000110, // R
        40'b01111110_00001001_00001001_00001001_01111110, // A
        40'b00100110_01001001_01001001_01001001_00110010, // S
        40'b00000000_01000001_01111111_01000001_00000000, // I
        40'b01111111_00000010_00000100_00001000_01111111, // N
        40'b00111110_01000001_01001001_01001001_00111010, // G
        40'b00000000_00000000_00000000_00000000_00000000, //
        40'b00000000_00000000_00000000_00000000_00000000, //
        40'b00000000_00000000_00000000_00000000_00000000, //
        address_dots};
       `STATUS_WRITING:
    dots <= {40'b01111111_00100000_00011000_00100000_01111111, // W
        40'b01111111_00001001_00011001_00101001_01000110, // R
        40'b00000000_01000001_01111111_01000001_00000000, // I
        40'b00000001_00000001_01111111_00000001_00000001, // T
        40'b00000000_01000001_01111111_01000001_00000000, // I
        40'b01111111_00000010_00000100_00001000_01111111, // N
        40'b00111110_01000001_01001001_01001001_00111010, // G
        40'b00000000_00000000_00000000_00000000_00000000, //
        40'b00000000_00000000_00000000_00000000_00000000, //
        40'b00000000_00000000_00000000_00000000_00000000, //
        address_dots};
       `STATUS_READING:
    dots <= {40'b01111111_00001001_00011001_00101001_01000110, // R
        40'b01111111_01001001_01001001_01001001_01000001, // E
        40'b01111110_00001001_00001001_00001001_01111110, // A
        40'b01111111_01000001_01000001_01000001_00111110, // D
        40'b00000000_01000001_01111111_01000001_00000000, // I
        40'b01111111_00000010_00000100_00001000_01111111, // N
        40'b00111110_01000001_01001001_01001001_00111010, // G
        40'b00000000_00000000_00000000_00000000_00000000, //
        40'b00000000_00000000_00000000_00000000_00000000, //
        40'b00000000_00000000_00000000_00000000_00000000, //
        address_dots};
       `STATUS_SUCCESS:
    dots <= {40'b00000000_00000000_00000000_00000000_00000000, //
        40'b00101010_00011100_01111111_00011100_00101010, // *
        40'b00101010_00011100_01111111_00011100_00101010, // *
        40'b00101010_00011100_01111111_00011100_00101010, // *
        40'b00000000_00000000_00000000_00000000_00000000, //
        40'b01111111_00001001_00001001_00001001_00000110, // P
        40'b01111110_00001001_00001001_00001001_01111110, // A
        40'b00100110_01001001_01001001_01001001_00110010, // S
        40'b00100110_01001001_01001001_01001001_00110010, // S
        40'b01111111_01001001_01001001_01001001_01000001, // E
        40'b01111111_01000001_01000001_01000001_00111110, // D
        40'b00000000_00000000_00000000_00000000_00000000, //
        40'b00101010_00011100_01111111_00011100_00101010, // *
        40'b00101010_00011100_01111111_00011100_00101010, // *
        40'b00101010_00011100_01111111_00011100_00101010, // *
        40'b00000000_00000000_00000000_00000000_00000000};//
       `STATUS_BAD_MANUFACTURER:
    dots <= {40'b01111111_01001001_01001001_01001001_01000001, // E
        40'b01111111_00001001_00011001_00101001_01000110, // R
        40'b01111111_00001001_00011001_00101001_01000110, // R
        40'b00000000_00110110_00110110_00000000_00000000, // :
        40'b00000000_00000000_00000000_00000000_00000000, //
        40'b01111111_00000010_00001100_00000010_01111111, // M
        40'b01111110_00001001_00001001_00001001_01111110, // A
        40'b01111111_00000010_00000100_00001000_01111111, // N
        40'b01111111_00001001_00001001_00001001_00000001, // U
        40'b01111111_00001001_00001001_00001001_00000001, // F
        40'b00000000_00000000_00000000_00000000_00000000, //
        40'b00000000_00000000_00000000_00000000_00000000, //
        data_dots};
       `STATUS_BAD_SIZE:
    dots <= {40'b01111111_01001001_01001001_01001001_01000001, // E
        40'b01111111_00001001_00011001_00101001_01000110, // R
        40'b01111111_00001001_00011001_00101001_01000110, // R
        40'b00000000_00110110_00110110_00000000_00000000, // :
        40'b00000000_00000000_00000000_00000000_00000000, //
        40'b00100110_01001001_01001001_01001001_00110010, // S
        40'b00000000_01000001_01111111_01000001_00000000, // I
        40'b01100001_01010001_01001001_01000101_01000011, // Z
        40'b01111111_01001001_01001001_01001001_01000001, // E
        40'b00000000_00000000_00000000_00000000_00000000,
        40'b00000000_00000000_00000000_00000000_00000000,
        40'b00000000_00000000_00000000_00000000_00000000,
        data_dots};
       `STATUS_LOCK_BIT_ERROR:
    dots <= {40'b01111111_01001001_01001001_01001001_01000001, // E
        40'b01111111_00001001_00011001_00101001_01000110, // R
        40'b01111111_00001001_00011001_00101001_01000110, // R
        40'b00000000_00110110_00110110_00000000_00000000, // :
        40'b00000000_00000000_00000000_00000000_00000000, //
        40'b01111111_01000000_01000000_01000000_01000000, // L
        40'b00111110_01000001_01000001_01000001_00111110, // O
        40'b00111110_01000001_01000001_01000001_00100010, // C
        40'b01111111_00001000_00010100_00100010_01000001, // K
        40'b00100110_01001001_01001001_01001001_00110010, // S
        40'b00000000_00000000_00000000_00000000_00000000,
        40'b00000000_00000000_00000000_00000000_00000000,
        data_dots};
       `STATUS_ERASE_BLOCK_ERROR:
    dots <= {40'b01111111_01001001_01001001_01001001_01000001, // E
        40'b01111111_00001001_00011001_00101001_01000110, // R
        40'b01111111_00001001_00011001_00101001_01000110, // R
        40'b00000000_00110110_00110110_00000000_00000000, // :
        40'b00000000_00000000_00000000_00000000_00000000, //
        40'b01111111_01001001_01001001_01001001_01000001, // E
        40'b01111111_00001001_00011001_00101001_01000110, // R
        40'b01111110_00001001_00001001_00001001_01111110, // A
        40'b00100110_01001001_01001001_01001001_00110010, // S
        40'b01111111_01001001_01001001_01001001_01000001, // E
        40'b00000000_00000000_00000000_00000000_00000000,
        40'b00000000_00000000_00000000_00000000_00000000,
        data_dots};
       `STATUS_WRITE_ERROR:
    dots <= {40'b01111111_01001001_01001001_01001001_01000001, // E
        40'b01111111_00001001_00011001_00101001_01000110, // R
        40'b01111111_00001001_00011001_00101001_01000110, // R
        40'b00000000_00110110_00110110_00000000_00000000, // :
        40'b00000000_00000000_00000000_00000000_00000000, //
        40'b01111111_00100000_00011000_00100000_01111111, // W
        40'b01111111_00001001_00011001_00101001_01000110, // R
        40'b00000000_01000001_01111111_01000001_00000000, // I
        40'b00000001_00000001_01111111_00000001_00000001, // T
        40'b01111111_01001001_01001001_01001001_01000001, // E
        40'b00000000_00000000_00000000_00000000_00000000,
        40'b00000000_00000000_00000000_00000000_00000000,
        data_dots};
       `STATUS_READ_WRONG_DATA:
    dots <= {40'b01111111_01001001_01001001_01001001_01000001, // E
        40'b01111111_00001001_00011001_00101001_01000110, // R
        40'b01111111_00001001_00011001_00101001_01000110, // R
        40'b00000000_00110110_00110110_00000000_00000000, // :
        40'b00000000_00000000_00000000_00000000_00000000,
        address_dots,
        40'b00000000_00000000_00000000_00000000_00000000,
        data_dots};
    default:
    dots <= {16{40'b01010101_00101010_01010101_00101010_01010101}};
     endcase

endmodule
```

```
//////////////////////////////////////////////////////////////////
//
// 6.111 FPGA Labkit -- Flash Tester State Machine
//
//
// Created: January 23, 2005
// Author: Nathan Ickes
//
//////////////////////////////////////////////////////////////////

`define FLASHOP_IDLE  2'b00
`define FLASHOP_READ  2'b01
`define FLASHOP_WRITE 2'b10

`define STATUS_RESET             4'h0
`define STATUS_READ_ID           4'h1
`define STATUS_CLEAR_LOCKS       4'h2
`define STATUS_ERASING           4'h3
`define STATUS_WRITING           4'h4
`define STATUS_READING           4'h5
`define STATUS_SUCCESS           4'h6
`define STATUS_BAD_MANUFACTURER  4'h7
`define STATUS_BAD_SIZE          4'h8
`define STATUS_LOCK_BIT_ERROR    4'h9
`define STATUS_ERASE_BLOCK_ERROR 4'hA
`define STATUS_WRITE_ERROR       4'hB
`define STATUS_READ_WRONG_DATA   4'hC

//`define NUM_BLOCKS 128
`define NUM_BLOCKS 128
`define BLOCK_SIZE 64*1024
`define LAST_BLOCK_ADDRESS ((`NUM_BLOCKS-1)*`BLOCK_SIZE)
`define LAST_ADDRESS (`NUM_BLOCKS*`BLOCK_SIZE-1)

`define NUM_BLOCKS2 2
`define BLOCK_SIZE2 64*1024
`define LAST_BLOCK_ADDRESS2 ((`NUM_BLOCKS2-1)*`BLOCK_SIZE2)
`define LAST_ADDRESS2 (`NUM_BLOCKS2*`BLOCK_SIZE2-1)


`define INIT_FLASH_ADDRESS   23'h0      // next one 23'd8192
`define LAST_FLASH_ADDRESS (`INIT_FLASH_ADDRESS+23'd8192)


module test_fsm (reset, clock, fop, faddress, fwdata, frdata, fbusy, dots, FPGA_ROM_data, FPGA_ROM_addr);

   input reset, clock;
   output [1:0] fop;
   output [22:0] faddress;
   output [15:0] fwdata;
   input [15:0]  frdata;
   input fbusy;
   output [639:0] dots;
   input [7:0] FPGA_ROM_data;
   output [12:0] FPGA_ROM_addr;

   reg [1:0] fop;
   reg [22:0] faddress;
   reg [15:0] fwdata;
   reg [639:0] dots;
   reg [12:0] FPGA_ROM_addr;

   ////////////////////////////////////////////////////////////////////////////
   //
   // State Machine
   //
   ////////////////////////////////////////////////////////////////////////////

   reg [7:0] state;
   reg [3:0] status;

   always @(posedge clock)
     if (reset)
       begin
     state <= 0;
     status <= `STATUS_RESET;
     faddress <= 0;
     FPGA_ROM_addr <= 0;
     fop <= `FLASHOP_IDLE;
```

```
       end
     else if (!fbusy && (fop == `FLASHOP_IDLE))
       case (state)
    8'h00:
      begin
         // Issue "read id codes" command
         status <= `STATUS_READ_ID;
         faddress <= 0;
         fwdata <= 16'h0090;
         fop <= `FLASHOP_WRITE;
         state <= state+1;
      end

    8'h01:
      begin
         // Read manufacturer code
         faddress <= 0;
         fop <= `FLASHOP_READ;
         state <= state+1;
      end

    8'h02:
      if (frdata != 16'h0089) // 16'h0089 = Intel
        status <= `STATUS_BAD_MANUFACTURER;
      else
        begin
      // Read the device size code
      faddress <= 1;
      fop <= `FLASHOP_READ;
      state <= state+1;
        end

    8'h03:
      if (frdata != 16'h0018) // 16'h0018 = 128Mbit
        status <= `STATUS_BAD_SIZE;
      else
        begin
      faddress <= 0;
      fwdata <= 16'hFF;
      fop <= `FLASHOP_WRITE;
      state <= state+1;
        end
    8'h04:
      begin
         // Issue "clear lock bits" command
         status <= `STATUS_CLEAR_LOCKS;
         faddress <= 0;
         fwdata <= 16'h60;
         fop <= `FLASHOP_WRITE;
         state <= state+1;
      end
    8'h05:
      begin
         // Issue "confirm clear lock bits" command
         faddress <= 0;
         fwdata <= 16'hD0;
         fop <= `FLASHOP_WRITE;
         state <= state+1;
      end
    8'h06:
      begin
         // Read status
         faddress <= 0;
         fop <= `FLASHOP_READ;
         state <= state+1;
      end
    8'h07:
      if (frdata[7] == 1) // Done clearing lock bits
        if (frdata[6:1] == 0) // No errors
          begin
        faddress <= 0;
        fop <= `FLASHOP_IDLE;
        state <= state+1;
          end
        else
          status <= `STATUS_LOCK_BIT_ERROR;
      else // Still busy, reread status register
        begin
      faddress <= 0;
      fop <= `FLASHOP_READ;
        end

    //////////////////////////////////////////////////////////////////////
    // Block Erase Sequence
    //////////////////////////////////////////////////////////////////////

    8'h08:
      begin
         status <= `STATUS_ERASING;
         fwdata <= 16'h20; // Issue "erase block" command
         fop <= `FLASHOP_WRITE;
         state <= state+1;
      end
    8'h09:
      begin
         fwdata <= 16'hD0; // Issue "confirm erase" command
         fop <= `FLASHOP_WRITE;
         state <= state+1;
      end
    8'h0A:
      begin
         fop <= `FLASHOP_READ;
         state <= state+1;
      end
    8'h0B:
      if (frdata[7] == 1) // Done erasing block
        if (frdata[6:1] == 0) // No errors
          if (faddress != 23'h7F0000) // `LAST_BLOCK_ADDRESS)
        //if (faddress != `LAST_BLOCK_ADDRESS) // `LAST_BLOCK_ADDRESS)
        //if (faddress != 23'h7F0000) // `LAST_BLOCK_ADDRESS)
        //if (faddress != 23'h100) // `LAST_BLOCK_ADDRESS)
       begin
          faddress <= faddress+`BLOCK_SIZE;
          fop <= `FLASHOP_IDLE;
          state <= state-3;
       end
          else
       begin
          faddress <= `INIT_FLASH_ADDRESS;
          FPGA_ROM_addr <= 0;
          fop <= `FLASHOP_IDLE;
          state <= state+1;
       end
        else // Erase error detected
          status <= `STATUS_ERASE_BLOCK_ERROR;
      else // Still busy
        fop <= `FLASHOP_READ;

    //////////////////////////////////////////////////////////////////////
    // Write Addresses to All Locations
    //////////////////////////////////////////////////////////////////////

    8'h0C:
      begin
         status <= `STATUS_WRITING;
         fwdata <= 16'h40; // Issue "setup write" command
         fop <= `FLASHOP_WRITE;
         state <= state+1;
      end

    8'h0D:
      begin
         //fwdata <= faddress[15:0]^faddress[22:16]; // Finish write
       //fwdata <= 16'b10101; // Finish write
       fwdata <= {8'b0, FPGA_ROM_data}; // Finish write
         fop <= `FLASHOP_WRITE;
         state <= state+1;
      end
    8'h0E:
      begin
         // Read status register
         fop <= `FLASHOP_READ;
         state <= state+1;
      end
    8'h0F:
      if (frdata[7] == 1) // Done writing
        if (frdata[6:1] == 0) // No errors
        if (faddress != `LAST_FLASH_ADDRESS) // `LAST_ADDRESS)
        //if (faddress != {20'b0, FPGA_ROM_addr}) // `LAST_ADDRESS)
        //if (faddress != 23'h7FFFFF) // `LAST_ADDRESS)  , if (faddress != 23'h20000)
       begin
          faddress <= faddress+1;
          FPGA_ROM_addr <= FPGA_ROM_addr+1;
          fop <= `FLASHOP_IDLE;
          state <= state-3;
       end
          else
       begin
          faddress <= `INIT_FLASH_ADDRESS;
          FPGA_ROM_addr <= 0;
          fop <= `FLASHOP_IDLE;
          state <= state+1;
       end
        else // Write error detected
          status <= `STATUS_WRITE_ERROR;
      else // Still busy
        fop <= `FLASHOP_READ;

    ////////////////////////////////////////////////////////////////////
    // Read back data
    ////////////////////////////////////////////////////////////////////

    8'h10:
      begin
         status <= `STATUS_READING;
         fwdata <= 16'hFF; // Issue "read array" command
         fop <= `FLASHOP_WRITE;
         faddress <= `INIT_FLASH_ADDRESS;
       FPGA_ROM_addr <= 0;
         state <= state+1;
      end
    8'h11:
      begin
         fop <= `FLASHOP_READ;
         state <= state+1;
      end
    8'h12:
      if (frdata == {8'b0, FPGA_ROM_data})
         if (faddress == `LAST_FLASH_ADDRESS)
      //if (faddress == `LAST_ADDRESS2)
      //if (faddress == 23'h7FFFFF)
          begin
        fop <= `FLASHOP_IDLE;
        faddress <= `INIT_FLASH_ADDRESS;
        FPGA_ROM_addr <= 0;
        state <= state+5;
          end
        else
          begin
        faddress <= faddress+1;
        FPGA_ROM_addr <= FPGA_ROM_addr+1;
        fop <= `FLASHOP_READ;
          end
      else
        status <= `STATUS_READ_WRONG_DATA;
  /*
    ////////////////////////////////////////////////////////////////////
    // Erase the chip again
    ////////////////////////////////////////////////////////////////////

    8'h13:
      begin
         status <= `STATUS_ERASING;
         fwdata <= 16'h20; // Issue "erase block" command
         fop <= `FLASHOP_WRITE;
         state <= state+1;
      end
    8'h14:
      begin
         fwdata <= 16'hD0; // Issue "confirm erase" command
         fop <= `FLASHOP_WRITE;
         state <= state+1;
      end
    8'h15:
      begin
         fop <= `FLASHOP_READ;
         state <= state+1;
      end
    8'h16:
      if (frdata[7] == 1) // Done erasing block
        if (frdata[6:1] == 0) // No errors
          if (faddress != 23'h20000) // `LAST_BLOCK_ADDRESS)
        //if (faddress != `LAST_BLOCK_ADDRESS) // `LAST_BLOCK_ADDRESS)
        // if (faddress != 23'h7F0000) // `LAST_BLOCK_ADDRESS)
        // if (faddress != 23'h100) // `LAST_BLOCK_ADDRESS)
       begin
          faddress <= faddress+`BLOCK_SIZE;
          fop <= `FLASHOP_IDLE;
          state <= state-3;
       end
          else
       begin
          faddress <= 0;
          fop <= `FLASHOP_IDLE;
          state <= state+1;
       end
        else // Erase error detected
          status <= `STATUS_ERASE_BLOCK_ERROR;
      else // Still busy
        fop <= `FLASHOP_READ;
*/
    ////////////////////////////////////////////////////////////////////////
    // End of test: declare success
    ////////////////////////////////////////////////////////////////////////

    8'h17:
      begin
         status <= `STATUS_SUCCESS;
         fop <= `FLASHOP_IDLE;
      end

       endcase
     else
       fop <= `FLASHOP_IDLE;

   ///////////////////////////////////////////////////////////////////////////
   //
   // Status display
   //
   ///////////////////////////////////////////////////////////////////////////
   //
   // "Reset     ------" --> During reset
   // "Read ID   000000" --> While reading ID codes
   // "Clr locks 000000" --> While clearing block locks
   // "Erase     000000" --> While erasing
   // "Write     000000" --> While writing
   // "Read      000000" --> While reading
   // " *** PASSED *** " --> If the entire test completes with no errors
   // "Err: Manuf  0000" --> If an incorrect manufacturer code is read
   // "Err: Size   0000" --> If an incorrect size code is read
   // "Err: Locks  0000" --> If an error is detected when clearing the block lock bits
   // "Err: Erase  0000"
   // "Err: 000000 0000"


   // Rd  000000  0000
   // Wr  000000  0000
   // Id  000000  0000
   //

   function [39:0] nib2char;
      input [3:0] nib;
      begin
    case (nib)
      4'h0: nib2char = 40'b00111110_01010001_01001001_01000101_00111110;
      4'h1: nib2char = 40'b00000000_01000010_01111111_01000000_00000000;
      4'h2: nib2char = 40'b01100010_01010001_01001001_01001001_01000110;
      4'h3: nib2char = 40'b00100010_01000001_01001001_01001001_00110110;
      4'h4: nib2char = 40'b00011000_00010100_00010010_01111111_00010000;
      4'h5: nib2char = 40'b00100111_01000101_01000101_01000101_00111001;
      4'h6: nib2char = 40'b00111100_01001010_01001001_01001001_00110000;
      4'h7: nib2char = 40'b00000001_01110001_00001001_00000101_00000011;
      4'h8: nib2char = 40'b00110110_01001001_01001001_01001001_00110110;
      4'h9: nib2char = 40'b00000110_01001001_01001001_00101001_00011110;
      4'hA: nib2char = 40'b01111110_00001001_00001001_00001001_01111110;
      4'hB: nib2char = 40'b01111111_01001001_01001001_01001001_00110110;
      4'hC: nib2char = 40'b00111110_01000001_01000001_01000001_00100010;
      4'hD: nib2char = 40'b01111111_01000001_01000001_01000001_00111110;
      4'hE: nib2char = 40'b01111111_01001001_01001001_01001001_01000001;
      4'hF: nib2char = 40'b01111111_00001001_00001001_00001001_00000001;
    endcase
      end
   endfunction

   wire [159:0] data_dots;
   assign data_dots = {nib2char(frdata[15:12]), nib2char(frdata[11:8]),
             nib2char(frdata[7:4]), nib2char(frdata[3:0])};

   wire [239:0] address_dots;
   assign address_dots = {nib2char({ 1'b0, faddress[22:20]}),
           nib2char(faddress[19:16]),
           nib2char(faddress[15:12]),
           nib2char(faddress[11:8]),
           nib2char(faddress[7:4]),
           nib2char(faddress[3:0])};

   always @(status or address_dots or data_dots)
     case (status)
       `STATUS_RESET:
    dots <= {40'b01111111_00001001_00011001_00101001_01000110, // R
        40'b01111111_01001001_01001001_01001001_01000001, // E
        40'b00100110_01001001_01001001_01001001_00110010, // S
        40'b01111111_01001001_01001001_01001001_01000001, // E
        40'b00000001_00000001_01111111_00000001_00000001, // T
        40'b00000000_00000000_00000000_00000000_00000000, //
        40'b00000000_00000000_00000000_00000000_00000000, //
        40'b00000000_00000000_00000000_00000000_00000000, //
        40'b00000000_00000000_00000000_00000000_00000000, //
        40'b00000000_00000000_00000000_00000000_00000000, //
        40'b00001000_00001000_00001000_00001000_00001000, // -
        40'b00001000_00001000_00001000_00001000_00001000, // -
        40'b00001000_00001000_00001000_00001000_00001000, // -
        40'b00001000_00001000_00001000_00001000_00001000, // -
        40'b00001000_00001000_00001000_00001000_00001000, // -
        40'b00001000_00001000_00001000_00001000_00001000};// -
       `STATUS_READ_ID:
    dots <= {40'b01111111_00001001_00011001_00101001_01000110, // R
        40'b01111111_01001001_01001001_01001001_01000001, // E
        40'b01111110_00001001_00001001_00001001_01111110, // A
        40'b01111111_01000001_01000001_01000001_00111110, // D
        40'b00000000_00000000_00000000_00000000_00000000, //
        40'b00000000_01000001_01111111_01000001_00000000, // I
        40'b01111111_01000001_01000001_01000001_00111110, // D
        40'b00000000_00000000_00000000_00000000_00000000, //
        40'b00000000_00000000_00000000_00000000_00000000, //
        40'b00000000_00000000_00000000_00000000_00000000, //
        address_dots};
       `STATUS_CLEAR_LOCKS:
    dots <= {40'b00111110_01000001_01000001_01000001_00100010, // C
        40'b01111111_01000000_01000000_01000000_01000000, // L
        40'b01111111_00001001_00011001_00101001_01000110, // R
        40'b00000000_00000000_00000000_00000000_00000000, //
        40'b01111111_01000000_01000000_01000000_01000000, // L
        40'b00111110_01000001_01000001_01000001_00111110, // O
        40'b00111110_01000001_01000001_01000001_00100010, // C
        40'b01111111_00001000_00010100_00100010_01000001, // K
        40'b00100110_01001001_01001001_01001001_00110010, // S
        40'b00000000_00000000_00000000_00000000_00000000, //
        address_dots};
       `STATUS_ERASING:
    dots <= {40'b01111111_01001001_01001001_01001001_01000001, // E
        40'b01111111_00001001_00011001_00101001_01000110, // R
        40'b01111110_00001001_00001001_00001001_01111110, // A
        40'b00100110_01001001_01001001_01001001_00110010, // S
        40'b00000000_01000001_01111111_01000001_00000000, // I
        40'b01111111_00000010_00000100_00001000_01111111, // N
        40'b00111110_01000001_01001001_01001001_00111010, // G
        40'b00000000_00000000_00000000_00000000_00000000, //
        40'b00000000_00000000_00000000_00000000_00000000, //
        40'b00000000_00000000_00000000_00000000_00000000, //
        address_dots};
       `STATUS_WRITING:
    dots <= {40'b01111111_00100000_00011000_00100000_01111111, // W
        40'b01111111_00001001_00011001_00101001_01000110, // R
        40'b00000000_01000001_01111111_01000001_00000000, // I
        40'b00000001_00000001_01111111_00000001_00000001, // T
        40'b00000000_01000001_01111111_01000001_00000000, // I
        40'b01111111_00000010_00000100_00001000_01111111, // N
        40'b00111110_01000001_01001001_01001001_00111010, // G
        40'b00000000_00000000_00000000_00000000_00000000, //
        40'b00000000_00000000_00000000_00000000_00000000, //
        40'b00000000_00000000_00000000_00000000_00000000, //
        address_dots};
       `STATUS_READING:
    dots <= {40'b01111111_00001001_00011001_00101001_01000110, // R
        40'b01111111_01001001_01001001_01001001_01000001, // E
        40'b01111110_00001001_00001001_00001001_01111110, // A
        40'b01111111_01000001_01000001_01000001_00111110, // D
        40'b00000000_01000001_01111111_01000001_00000000, // I
        40'b01111111_00000010_00000100_00001000_01111111, // N
        40'b00111110_01000001_01001001_01001001_00111010, // G
        40'b00000000_00000000_00000000_00000000_00000000, //
        40'b00000000_00000000_00000000_00000000_00000000, //
        40'b00000000_00000000_00000000_00000000_00000000, //
        address_dots};
       `STATUS_SUCCESS:
    dots <= {40'b00000000_00000000_00000000_00000000_00000000, //
        40'b00101010_00011100_01111111_00011100_00101010, // *
        40'b00101010_00011100_01111111_00011100_00101010, // *
        40'b00101010_00011100_01111111_00011100_00101010, // *
        40'b00000000_00000000_00000000_00000000_00000000, //
        40'b01111111_00001001_00001001_00001001_00000110, // P
        40'b01111110_00001001_00001001_00001001_01111110, // A
        40'b00100110_01001001_01001001_01001001_00110010, // S
        40'b00100110_01001001_01001001_01001001_00110010, // S
        40'b01111111_01001001_01001001_01001001_01000001, // E
        40'b01111111_01000001_01000001_01000001_00111110, // D
        40'b00000000_00000000_00000000_00000000_00000000, //
        40'b00101010_00011100_01111111_00011100_00101010, // *
        40'b00101010_00011100_01111111_00011100_00101010, // *
        40'b00101010_00011100_01111111_00011100_00101010, // *
        40'b00000000_00000000_00000000_00000000_00000000};//
       `STATUS_BAD_MANUFACTURER:
    dots <= {40'b01111111_01001001_01001001_01001001_01000001, // E
        40'b01111111_00001001_00011001_00101001_01000110, // R
        40'b01111111_00001001_00011001_00101001_01000110, // R
        40'b00000000_00110110_00110110_00000000_00000000, // :
        40'b00000000_00000000_00000000_00000000_00000000, //
        40'b01111111_00000010_00001100_00000010_01111111, // M
        40'b01111110_00001001_00001001_00001001_01111110, // A
        40'b01111111_00000010_00000100_00001000_01111111, // N
        40'b01111111_00001001_00001001_00001001_00000001, // U
        40'b01111111_00001001_00001001_00001001_00000001, // F
        40'b00000000_00000000_00000000_00000000_00000000, //
        40'b00000000_00000000_00000000_00000000_00000000, //
        data_dots};
       `STATUS_BAD_SIZE:
    dots <= {40'b01111111_01001001_01001001_01001001_01000001, // E
        40'b01111111_00001001_00011001_00101001_01000110, // R
        40'b01111111_00001001_00011001_00101001_01000110, // R
        40'b00000000_00110110_00110110_00000000_00000000, // :
        40'b00000000_00000000_00000000_00000000_00000000, //
        40'b00100110_01001001_01001001_01001001_00110010, // S
        40'b00000000_01000001_01111111_01000001_00000000, // I
        40'b01100001_01010001_01001001_01000101_01000011, // Z
        40'b01111111_01001001_01001001_01001001_01000001, // E
        40'b00000000_00000000_00000000_00000000_00000000,
        40'b00000000_00000000_00000000_00000000_00000000,
        40'b00000000_00000000_00000000_00000000_00000000,
        data_dots};
       `STATUS_LOCK_BIT_ERROR:
    dots <= {40'b01111111_01001001_01001001_01001001_01000001, // E
        40'b01111111_00001001_00011001_00101001_01000110, // R
        40'b01111111_00001001_00011001_00101001_01000110, // R
        40'b00000000_00110110_00110110_00000000_00000000, // :
        40'b00000000_00000000_00000000_00000000_00000000, //
        40'b01111111_01000000_01000000_01000000_01000000, // L
        40'b00111110_01000001_01000001_01000001_00111110, // O
        40'b00111110_01000001_01000001_01000001_00100010, // C
        40'b01111111_00001000_00010100_00100010_01000001, // K
        40'b00100110_01001001_01001001_01001001_00110010, // S
        40'b00000000_00000000_00000000_00000000_00000000,
        40'b00000000_00000000_00000000_00000000_00000000,
        data_dots};
       `STATUS_ERASE_BLOCK_ERROR:
    dots <= {40'b01111111_01001001_01001001_01001001_01000001, // E
        40'b01111111_00001001_00011001_00101001_01000110, // R
        40'b01111111_00001001_00011001_00101001_01000110, // R
        40'b00000000_00110110_00110110_00000000_00000000, // :
        40'b00000000_00000000_00000000_00000000_00000000, //
        40'b01111111_01001001_01001001_01001001_01000001, // E
        40'b01111111_00001001_00011001_00101001_01000110, // R
        40'b01111110_00001001_00001001_00001001_01111110, // A
        40'b00100110_01001001_01001001_01001001_00110010, // S
        40'b01111111_01001001_01001001_01001001_01000001, // E
        40'b00000000_00000000_00000000_00000000_00000000,
        40'b00000000_00000000_00000000_00000000_00000000,
        data_dots};
       `STATUS_WRITE_ERROR:
    dots <= {40'b01111111_01001001_01001001_01001001_01000001, // E
        40'b01111111_00001001_00011001_00101001_01000110, // R
        40'b01111111_00001001_00011001_00101001_01000110, // R
        40'b00000000_00110110_00110110_00000000_00000000, // :
        40'b00000000_00000000_00000000_00000000_00000000, //
        40'b01111111_00100000_00011000_00100000_01111111, // W
        40'b01111111_00001001_00011001_00101001_01000110, // R
        40'b00000000_01000001_01111111_01000001_00000000, // I
        40'b00000001_00000001_01111111_00000001_00000001, // T
        40'b01111111_01001001_01001001_01001001_01000001, // E
        40'b00000000_00000000_00000000_00000000_00000000,
        40'b00000000_00000000_00000000_00000000_00000000,
        data_dots};
       `STATUS_READ_WRONG_DATA:
    dots <= {40'b01111111_01001001_01001001_01001001_01000001, // E
        40'b01111111_00001001_00011001_00101001_01000110, // R
        40'b01111111_00001001_00011001_00101001_01000110, // R
        40'b00000000_00110110_00110110_00000000_00000000, // :
        40'b00000000_00000000_00000000_00000000_00000000,
        address_dots,
        40'b00000000_00000000_00000000_00000000_00000000,
        data_dots};
    default:
    dots <= {16{40'b01010101_00101010_01010101_00101010_01010101}};
     endcase

endmodule
```

```
///////////////////////////////////////////////////////////////////
//
// 6.111 FPGA Labkit -- Flash Tester State Machine
//
//
// Created: January 23, 2005
// Author: Nathan Ickes
//
///////////////////////////////////////////////////////////////////

`define FLASHOP_IDLE  2'b00
`define FLASHOP_READ  2'b01
`define FLASHOP_WRITE 2'b10

`define STATUS_RESET             4'h0
`define STATUS_READ_ID           4'h1
`define STATUS_CLEAR_LOCKS       4'h2
`define STATUS_ERASING           4'h3
`define STATUS_WRITING           4'h4
`define STATUS_READING           4'h5
`define STATUS_SUCCESS           4'h6
`define STATUS_BAD_MANUFACTURER  4'h7
`define STATUS_BAD_SIZE          4'h8
`define STATUS_LOCK_BIT_ERROR    4'h9
`define STATUS_ERASE_BLOCK_ERROR 4'hA
`define STATUS_WRITE_ERROR       4'hB
`define STATUS_READ_WRONG_DATA   4'hC
`define STATUS_READOUT_DATA      4'hD

//`define NUM_BLOCKS 128
`define NUM_BLOCKS 128
`define BLOCK_SIZE 64*1024
`define LAST_BLOCK_ADDRESS ((`NUM_BLOCKS-1)*`BLOCK_SIZE)
`define LAST_ADDRESS (`NUM_BLOCKS*`BLOCK_SIZE-1)

`define NUM_BLOCKS2 2
`define BLOCK_SIZE2 64*1024
`define LAST_BLOCK_ADDRESS2 ((`NUM_BLOCKS2-1)*`BLOCK_SIZE2)
`define LAST_ADDRESS2 (`NUM_BLOCKS2*`BLOCK_SIZE2-1)


`define INIT_FLASH_ADDRESS   23'd32772     // 23'h0, 23'd8192
`define LAST_FLASH_ADDRESS (`INIT_FLASH_ADDRESS+23'd20)


module test_fsm (reset, clock, fop, faddress, fwdata, frdata, fbusy, dots, FPGA_ROM_data, FPGA_ROM_addr);

   input reset, clock;
   output [1:0] fop;
   output [22:0] faddress;
   output [15:0] fwdata;
   input [15:0]  frdata;
   input fbusy;
   output [639:0] dots;
   input [7:0] FPGA_ROM_data;
   output [12:0] FPGA_ROM_addr;

   reg [1:0] fop;
   reg [22:0] faddress;
   reg [15:0] fwdata;
   reg [639:0] dots;
   reg [12:0] FPGA_ROM_addr;
   reg [25:0] count;

   ////////////////////////////////////////////////////////////////////////////
   //
   // State Machine
   //
   ////////////////////////////////////////////////////////////////////////////

   reg [7:0] state;
   reg [3:0] status;

   always @(posedge clock)
     if (reset)
       begin
     state <= 0;
     status <= `STATUS_RESET;
     faddress <= 0;
     FPGA_ROM_addr <= 0;
     count <= 0;
     fop <= `FLASHOP_IDLE;
       end
     else if (!fbusy && (fop == `FLASHOP_IDLE))
       case (state)
    8'h00:
      begin
         // Issue "read id codes" command
         status <= `STATUS_READ_ID;
         faddress <= 0;
         fwdata <= 16'h0090;
         fop <= `FLASHOP_WRITE;
         state <= state+1;
      end

    8'h01:
      begin
         // Read manufacturer code
         faddress <= 0;
         fop <= `FLASHOP_READ;
         state <= state+1;
      end

    8'h02:
      if (frdata != 16'h0089) // 16'h0089 = Intel
        status <= `STATUS_BAD_MANUFACTURER;
      else
        begin
      // Read the device size code
      faddress <= 1;
      fop <= `FLASHOP_READ;
      state <= state+1;
        end

    8'h03:
      if (frdata != 16'h0018) // 16'h0018 = 128Mbit
        status <= `STATUS_BAD_SIZE;
      else
        begin
      faddress <= 0;
      fwdata <= 16'hFF;
      fop <= `FLASHOP_WRITE;
      state <= state+1;
        end
    8'h04:
      begin
         // Issue "clear lock bits" command
         status <= `STATUS_CLEAR_LOCKS;
         faddress <= 0;
         fwdata <= 16'h60;
         fop <= `FLASHOP_WRITE;
         state <= state+1;
      end
    8'h05:
      begin
         // Issue "confirm clear lock bits" command
         faddress <= 0;
         fwdata <= 16'hD0;
         fop <= `FLASHOP_WRITE;
         state <= state+1;
      end
    8'h06:
      begin
         // Read status
         faddress <= 0;
         fop <= `FLASHOP_READ;
         state <= state+1;
      end
    8'h07:
      if (frdata[7] == 1) // Done clearing lock bits
        if (frdata[6:1] == 0) // No errors
          begin
        faddress <= `INIT_FLASH_ADDRESS;    // `INIT_FLASH_ADDRESS
        fop <= `FLASHOP_IDLE;
        state <= state+ 8'd9;
          end
        else
          status <= `STATUS_LOCK_BIT_ERROR;
      else // Still busy, reread status register
        begin
      faddress <= 0;
      fop <= `FLASHOP_READ;
        end
/*
    ///////////////////////////////////////////////////////////////////////
    // Block Erase Sequence
    ///////////////////////////////////////////////////////////////////////

    8'h08:
      begin
         status <= `STATUS_ERASING;
         fwdata <= 16'h20; // Issue "erase block" command
         fop <= `FLASHOP_WRITE;
         state <= state+1;
      end
    8'h09:
      begin
         fwdata <= 16'hD0; // Issue "confirm erase" command
         fop <= `FLASHOP_WRITE;
         state <= state+1;
      end
    8'h0A:
      begin
         fop <= `FLASHOP_READ;
         state <= state+1;
      end
    8'h0B:
      if (frdata[7] == 1) // Done erasing block
        if (frdata[6:1] == 0) // No errors
          if (faddress != 23'h7F0000) // `LAST_BLOCK_ADDRESS)
        //if (faddress != `LAST_BLOCK_ADDRESS) // `LAST_BLOCK_ADDRESS)
        //if (faddress != 23'h7F0000) // `LAST_BLOCK_ADDRESS)
        //if (faddress != 23'h100) // `LAST_BLOCK_ADDRESS)
       begin
          faddress <= faddress+`BLOCK_SIZE;
          fop <= `FLASHOP_IDLE;
          state <= state-3;
       end
          else
       begin
          faddress <= `INIT_FLASH_ADDRESS;
          FPGA_ROM_addr <= 0;
          fop <= `FLASHOP_IDLE;
          state <= state+1;
       end
        else // Erase error detected
          status <= `STATUS_ERASE_BLOCK_ERROR;
      else // Still busy
        fop <= `FLASHOP_READ;

*/
    ///////////////////////////////////////////////////////////////////////
    // Write Addresses to All Locations
    ///////////////////////////////////////////////////////////////////////
/*
    8'h0C:
      begin
         status <= `STATUS_WRITING;
         fwdata <= 16'h40; // Issue "setup write" command
         fop <= `FLASHOP_WRITE;
         state <= state+1;
      end

    8'h0D:
      begin
       //fwdata <= 16'b10101; // Finish write
       fwdata <= {8'b0, FPGA_ROM_data}; // Finish write
         fop <= `FLASHOP_WRITE;
         state <= state+1;
      end
    8'h0E:
      begin
         // Read status register
         fop <= `FLASHOP_READ;
         state <= state+1;
      end
    8'h0F:
```

```
[redacted]
*/
    //////////////////////////////////////////////////////////////////////
    // Read back data
    //////////////////////////////////////////////////////////////////////

    8'h10:
      begin
         status <= `STATUS_READING;
         fwdata <= 16'hFF; // Issue "read array" command
         fop <= `FLASHOP_WRITE;
         faddress <= `INIT_FLASH_ADDRESS;
       FPGA_ROM_addr <= 0;
         state <= state+1;
      end
    8'h11:
      begin
         fop <= `FLASHOP_READ;
         state <= state+1;
       status <= `STATUS_READOUT_DATA;
      end
    8'h12:
      begin
          fop <= `FLASHOP_IDLE;
       count <= count + 1;
       //status <= `STATUS_READOUT_DATA;
       state <= state+1;
        end
   8'h13:
      begin
       fop <= `FLASHOP_IDLE;
       //status <= `STATUS_READOUT_DATA;
       if (count == 26'd30000000)
         state <= state+1;
       else
         state <= state-1;
      end
    8'h14:
        if (faddress == `LAST_FLASH_ADDRESS)
      //if (faddress == 23'h7FFFFF)
          begin
        fop <= `FLASHOP_IDLE;
        faddress <= `INIT_FLASH_ADDRESS;
        FPGA_ROM_addr <= 0;
        state <= state+5;
          end
        else
          begin
        faddress <= faddress+1;
        FPGA_ROM_addr <= FPGA_ROM_addr+1;
        count <= 0;
        fop <= `FLASHOP_READ;
        state <= state-2;
        status <= `STATUS_READOUT_DATA;
          end

/*    if (frdata == {8'b0, FPGA_ROM_data})
         if (faddress == `LAST_FLASH_ADDRESS)
```

```
[redacted]
*/
  /*
    ///////////////////////////////////////////////////////////////////////
    // Erase the chip again
    ///////////////////////////////////////////////////////////////////////

    8'h13:
      begin
         status <= `STATUS_ERASING;
         fwdata <= 16'h20; // Issue "erase block" command
         fop <= `FLASHOP_WRITE;
         state <= state+1;
      end
    8'h14:
      begin
         fwdata <= 16'hD0; // Issue "confirm erase" command
         fop <= `FLASHOP_WRITE;
         state <= state+1;
      end
    8'h15:
      begin
         fop <= `FLASHOP_READ;
         state <= state+1;
      end
    8'h16:
      if (frdata[7] == 1) // Done erasing block
        if (frdata[6:1] == 0) // No errors
          if (faddress != 23'h20000) // `LAST_BLOCK_ADDRESS)
        //if (faddress != `LAST_BLOCK_ADDRESS) // `LAST_BLOCK_ADDRESS)
        // if (faddress != 23'h7F0000) // `LAST_BLOCK_ADDRESS)
        // if (faddress != 23'h100) // `LAST_BLOCK_ADDRESS)
       begin
          faddress <= faddress+`BLOCK_SIZE;
          fop <= `FLASHOP_IDLE;
          state <= state-3;
       end
          else
       begin
          faddress <= 0;
          fop <= `FLASHOP_IDLE;
          state <= state+1;
       end
        else // Erase error detected
          status <= `STATUS_ERASE_BLOCK_ERROR;
      else // Still busy
        fop <= `FLASHOP_READ;
*/
    ///////////////////////////////////////////////////////////////////////
    // End of test: declare success
    ///////////////////////////////////////////////////////////////////////

    8'h19:
      begin
         status <= `STATUS_SUCCESS;
         fop <= `FLASHOP_IDLE;
      end

       endcase
     else
       fop <= `FLASHOP_IDLE;

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

```verilog
//
// Status display
//
////////////////////////////////////////////////////////////////////////////
//
// "Reset     ------" --> During reset
// "Read ID   000000" --> While reading ID codes
// "Clr locks 000000" --> While clearing block locks
// "Erase     000000" --> While erasing
// "Write     000000" --> While writing
// "Read      000000" --> While reading
// " *** PASSED *** " --> If the entire test completes with no errors
// "Err: Manuf  0000" --> If an incorrect manufacturer code is read
// "Err: Size   0000" --> If an incorrect size code is read
// "Err: Locks  0000" --> If an error is detected when clearing the block lock bits
// "Err: Erase  0000"
// "Err: 000000 0000"


// Rd  000000  0000
// Wr  000000  0000
// Id  000000  0000
//

function [39:0] nib2char;
   input [3:0] nib;
   begin
 case (nib)
   4'h0: nib2char = 40'b00111110_01010001_01001001_01000101_00111110;
   4'h1: nib2char = 40'b00000000_01000010_01111111_01000000_00000000;
   4'h2: nib2char = 40'b01100010_01010001_01001001_01001001_01000110;
   4'h3: nib2char = 40'b00100010_01000001_01001001_01001001_00110110;
   4'h4: nib2char = 40'b00011000_00010100_00010010_01111111_00010000;
   4'h5: nib2char = 40'b00100111_01000101_01000101_01000101_00111001;
   4'h6: nib2char = 40'b00111100_01001010_01001001_01001001_00110000;
   4'h7: nib2char = 40'b00000001_01110001_00001001_00000101_00000011;
   4'h8: nib2char = 40'b00110110_01001001_01001001_01001001_00110110;
   4'h9: nib2char = 40'b00000110_01001001_01001001_00101001_00011110;
   4'hA: nib2char = 40'b01111110_00001001_00001001_00001001_01111110;
   4'hB: nib2char = 40'b01111111_01001001_01001001_01001001_00110110;
   4'hC: nib2char = 40'b00111110_01000001_01000001_01000001_00100010;
   4'hD: nib2char = 40'b01111111_01000001_01000001_01000001_00111110;
   4'hE: nib2char = 40'b01111111_01001001_01001001_01001001_01000001;
   4'hF: nib2char = 40'b01111111_00001001_00001001_00001001_00000001;
 endcase
   end
endfunction

wire [159:0] data_dots;
assign data_dots = {nib2char(frdata[15:12]), nib2char(frdata[11:8]),
          nib2char(frdata[7:4]), nib2char(frdata[3:0])};

wire [239:0] address_dots;
assign address_dots = {nib2char({ 1'b0, faddress[22:20]}),
        nib2char(faddress[19:16]),
        nib2char(faddress[15:12]),
        nib2char(faddress[11:8]),
        nib2char(faddress[7:4]),
        nib2char(faddress[3:0])};

always @(status or address_dots or data_dots)
  case (status)
    `STATUS_RESET:
 dots <= {40'b01111111_00001001_00011001_00101001_01000110, // R
     40'b01111111_01001001_01001001_01001001_01000001, // E
     40'b00100110_01001001_01001001_01001001_00110010, // S
     40'b01111111_01001001_01001001_01001001_01000001, // E
     40'b00000001_00000001_01111111_00000001_00000001, // T
     40'b00000000_00000000_00000000_00000000_00000000, //
     40'b00000000_00000000_00000000_00000000_00000000, //
     40'b00000000_00000000_00000000_00000000_00000000, //
     40'b00000000_00000000_00000000_00000000_00000000, //
     40'b00000000_00000000_00000000_00000000_00000000, //
     40'b00001000_00001000_00001000_00001000_00001000, // -
     40'b00001000_00001000_00001000_00001000_00001000, // -
     40'b00001000_00001000_00001000_00001000_00001000, // -
     40'b00001000_00001000_00001000_00001000_00001000, // -
     40'b00001000_00001000_00001000_00001000_00001000, // -
     40'b00001000_00001000_00001000_00001000_00001000};// -
```

```
       `STATUS_READ_ID:
    dots <= {40'b01111111_00001001_00011001_00101001_01000110, // R
        40'b01111111_01001001_01001001_01001001_01000001, // E
        40'b01111110_00001001_00001001_00001001_01111110, // A
        40'b01111111_01000001_01000001_01000001_00111110, // D
        40'b00000000_00000000_00000000_00000000_00000000, //
        40'b00000000_01000001_01111111_01000001_00000000, // I
        40'b01111111_01000001_01000001_01000001_00111110, // D
        40'b00000000_00000000_00000000_00000000_00000000, //
        40'b00000000_00000000_00000000_00000000_00000000, //
        40'b00000000_00000000_00000000_00000000_00000000, //
        address_dots};
       `STATUS_CLEAR_LOCKS:
    dots <= {40'b00111110_01000001_01000001_01000001_00100010, // C
        40'b01111111_01000000_01000000_01000000_01000000, // L
        40'b01111111_00001001_00011001_00101001_01000110, // R
        40'b00000000_00000000_00000000_00000000_00000000, //
        40'b01111111_01000000_01000000_01000000_01000000, // L
        40'b00111110_01000001_01000001_01000001_00111110, // O
        40'b00111110_01000001_01000001_01000001_00100010, // C
        40'b01111111_00001000_00010100_00100010_01000001, // K
        40'b00100110_01001001_01001001_01001001_00110010, // S
        40'b00000000_00000000_00000000_00000000_00000000, //
        address_dots};
       `STATUS_ERASING:
    dots <= {40'b01111111_01001001_01001001_01001001_01000001, // E
        40'b01111111_00001001_00011001_00101001_01000110, // R
        40'b01111110_00001001_00001001_00001001_01111110, // A
        40'b00100110_01001001_01001001_01001001_00110010, // S
        40'b00000000_01000001_01111111_01000001_00000000, // I
        40'b01111111_00000010_00000100_00001000_01111111, // N
        40'b00111110_01000001_01001001_01001001_00111010, // G
        40'b00000000_00000000_00000000_00000000_00000000, //
        40'b00000000_00000000_00000000_00000000_00000000, //
        40'b00000000_00000000_00000000_00000000_00000000, //
        address_dots};
       `STATUS_WRITING:
    dots <= {40'b01111111_00100000_00011000_00100000_01111111, // W
        40'b01111111_00001001_00011001_00101001_01000110, // R
        40'b00000000_01000001_01111111_01000001_00000000, // I
        40'b00000001_00000001_01111111_00000001_00000001, // T
        40'b00000000_01000001_01111111_01000001_00000000, // I
        40'b01111111_00000010_00000100_00001000_01111111, // N
        40'b00111110_01000001_01001001_01001001_00111010, // G
        40'b00000000_00000000_00000000_00000000_00000000, //
        40'b00000000_00000000_00000000_00000000_00000000, //
        40'b00000000_00000000_00000000_00000000_00000000, //
        address_dots};
       `STATUS_READING:
    dots <= {40'b01111111_00001001_00011001_00101001_01000110, // R
        40'b01111111_01001001_01001001_01001001_01000001, // E
        40'b01111110_00001001_00001001_00001001_01111110, // A
        40'b01111111_01000001_01000001_01000001_00111110, // D
        40'b00000000_01000001_01111111_01000001_00000000, // I
        40'b01111111_00000010_00000100_00001000_01111111, // N
        40'b00111110_01000001_01001001_01001001_00111010, // G
        40'b00000000_00000000_00000000_00000000_00000000, //
        40'b00000000_00000000_00000000_00000000_00000000, //
        40'b00000000_00000000_00000000_00000000_00000000, //
        address_dots};
       `STATUS_SUCCESS:
    dots <= {40'b00000000_00000000_00000000_00000000_00000000, //
        40'b00101010_00011100_01111111_00011100_00101010, // *
        40'b00101010_00011100_01111111_00011100_00101010, // *
        40'b00101010_00011100_01111111_00011100_00101010, // *
        40'b00000000_00000000_00000000_00000000_00000000, //
        40'b01111111_00001001_00001001_00001001_00000110, // P
        40'b01111110_00001001_00001001_00001001_01111110, // A
        40'b00100110_01001001_01001001_01001001_00110010, // S
        40'b00100110_01001001_01001001_01001001_00110010, // S
        40'b01111111_01001001_01001001_01001001_01000001, // E
        40'b01111111_01000001_01000001_01000001_00111110, // D
        40'b00000000_00000000_00000000_00000000_00000000, //
        40'b00101010_00011100_01111111_00011100_00101010, // *
        40'b00101010_00011100_01111111_00011100_00101010, // *
        40'b00101010_00011100_01111111_00011100_00101010, // *
        40'b00000000_00000000_00000000_00000000_00000000};//
       `STATUS_BAD_MANUFACTURER:
    dots <= {40'b01111111_01001001_01001001_01001001_01000001, // E
```

```
             40'b01111111_00001001_00011001_00101001_01000110, // R
             40'b01111111_00001001_00011001_00101001_01000110, // R
             40'b00000000_00110110_00110110_00000000_00000000, // :
             40'b00000000_00000000_00000000_00000000_00000000, //
             40'b01111111_00000010_00001100_00000010_01111111, // M
             40'b01111110_00001001_00001001_00001001_01111110, // A
             40'b01111111_00000010_00000100_00001000_01111111, // N
             40'b01111111_00001001_00001001_00001001_00000001, // U
             40'b01111111_00001001_00001001_00001001_00000001, // F
             40'b00000000_00000000_00000000_00000000_00000000, //
             40'b00000000_00000000_00000000_00000000_00000000, //
             data_dots};
            `STATUS_BAD_SIZE:
         dots <= {40'b01111111_01001001_01001001_01001001_01000001, // E
             40'b01111111_00001001_00011001_00101001_01000110, // R
             40'b01111111_00001001_00011001_00101001_01000110, // R
             40'b00000000_00110110_00110110_00000000_00000000, // :
             40'b00000000_00000000_00000000_00000000_00000000, //
             40'b00100110_01001001_01001001_01001001_00110010, // S
             40'b00000000_01000001_01111111_01000001_00000000, // I
             40'b01100001_01010001_01001001_01000101_01000011, // Z
             40'b01111111_01001001_01001001_01001001_01000001, // E
             40'b00000000_00000000_00000000_00000000_00000000,
             40'b00000000_00000000_00000000_00000000_00000000,
             40'b00000000_00000000_00000000_00000000_00000000,
             data_dots};
            `STATUS_LOCK_BIT_ERROR:
         dots <= {40'b01111111_01001001_01001001_01001001_01000001, // E
             40'b01111111_00001001_00011001_00101001_01000110, // R
             40'b01111111_00001001_00011001_00101001_01000110, // R
             40'b00000000_00110110_00110110_00000000_00000000, // :
             40'b00000000_00000000_00000000_00000000_00000000, //
             40'b01111111_01000000_01000000_01000000_01000000, // L
             40'b00111110_01000001_01000001_01000001_00111110, // O
             40'b00111110_01000001_01000001_01000001_00100010, // C
             40'b01111111_00001000_00010100_00100010_01000001, // K
             40'b00100110_01001001_01001001_01001001_00110010, // S
             40'b00000000_00000000_00000000_00000000_00000000,
             40'b00000000_00000000_00000000_00000000_00000000,
             data_dots};
            `STATUS_ERASE_BLOCK_ERROR:
         dots <= {40'b01111111_01001001_01001001_01001001_01000001, // E
             40'b01111111_00001001_00011001_00101001_01000110, // R
             40'b01111111_00001001_00011001_00101001_01000110, // R
             40'b00000000_00110110_00110110_00000000_00000000, // :
             40'b00000000_00000000_00000000_00000000_00000000, //
             40'b01111111_01001001_01001001_01001001_01000001, // E
             40'b01111111_00001001_00011001_00101001_01000110, // R
             40'b01111110_00001001_00001001_00001001_01111110, // A
             40'b00100110_01001001_01001001_01001001_00110010, // S
             40'b01111111_01001001_01001001_01001001_01000001, // E
             40'b00000000_00000000_00000000_00000000_00000000,
             40'b00000000_00000000_00000000_00000000_00000000,
             data_dots};
            `STATUS_WRITE_ERROR:
         dots <= {40'b01111111_01001001_01001001_01001001_01000001, // E
             40'b01111111_00001001_00011001_00101001_01000110, // R
             40'b01111111_00001001_00011001_00101001_01000110, // R
             40'b00000000_00110110_00110110_00000000_00000000, // :
             40'b00000000_00000000_00000000_00000000_00000000, //
             40'b01111111_00100000_00011000_00100000_01111111, // W
             40'b01111111_00001001_00011001_00101001_01000110, // R
             40'b00000000_01000001_01111111_01000001_00000000, // I
             40'b00000001_00000001_01111111_00000001_00000001, // T
             40'b01111111_01001001_01001001_01001001_01000001, // E
             40'b00000000_00000000_00000000_00000000_00000000,
             40'b00000000_00000000_00000000_00000000_00000000,
             data_dots};
            `STATUS_READ_WRONG_DATA:
         dots <= {40'b01111111_01001001_01001001_01001001_01000001, // E
             40'b01111111_00001001_00011001_00101001_01000110, // R
             40'b01111111_00001001_00011001_00101001_01000110, // R
             40'b00000000_00110110_00110110_00000000_00000000, // :
             40'b00000000_00000000_00000000_00000000_00000000,
             address_dots,
             40'b00000000_00000000_00000000_00000000_00000000,
             data_dots};
        `STATUS_READOUT_DATA:
         dots <= {40'b01111111_00001001_00011001_00101001_01000110, // R
             40'b01111111_01001001_01001001_01001001_01000001, // E
             40'b01111110_00001001_00001001_00001001_01111110, // A
             40'b01111111_01000001_01000001_01000001_00111110, // D
             40'b00000000_00000000_00000000_00000000_00000000, //
             address_dots,                       // 6 digits
             40'b00000000_00000000_00000000_00000000_00000000, //
             data_dots};                            // 4 digits
         default:
         dots <= {16{40'b01010101_00101010_01010101_00101010_01010101}};
          endcase

     endmodule
```

```verilog
module top(clk, reset, status, cs_bar, r_wbar, cs_bar_DA, ext_data, heartbeat, heartrate, digit2, digit1, digit0, abnormal);

input clk, reset, status;
inout[7:0] ext_data;
output cs_bar, r_wbar, cs_bar_DA, heartbeat, abnormal;
output[3:0] digit2, digit1, digit0;
output[8:0] heartrate;

wire reset_sync, status_sync, sample, sram_we, heartbeat, convert, abnormal;
wire filter_start, filter_busy, DAC_start, analog_busy, ADC_start;
wire[7:0] sram_q, rom1_q, rom2_q, read_data, int_data;
wire[3:0] sram_addr, rom1_addr, rom2_addr;
wire[8:0] heartrate;

synchronizer synchronizer1(clk, reset, status, reset_sync, status_sync);
enable250Hz enable250Hz1(clk, reset_sync, sample);
majorFSM majorFSM1(clk, reset_sync, sample, filter_busy, analog_busy, filter_start, DAC_start);
rom1 rom11(rom1_addr, rom1_q);
rom2 rom22(rom2_addr, rom2_q);
sram sram1(sram_addr, clk, read_data, sram_we, sram_q);
analog analog1(clk, reset_sync, ADC_start, DAC_start, status_sync, int_data, analog_busy, r_wbar, cs_bar, cs_bar_DA, ext_data, read_data);
filter filter1(clk, reset_sync, filter_start, sram_q, rom1_q, rom2_q, filter_busy, ADC_start, sram_we, sram_addr, rom1_addr, rom2_addr, int_data);
heartbeat_detector heartbeat_detector1(clk, reset_sync, int_data, heartbeat);
heartrate_detector heartrate_detector1(clk, reset_sync, heartbeat, heartrate, convert);
bin2dec bin2dec1(clk, reset_sync, convert, heartrate, digit2, digit1, digit0);
abnormality_detector abnormality_detector1(clk, reset_sync, heartrate, abnormal);

endmodule
```

```
module wireless_analog(clk, reset_sync, sample_wireless, cs_bar_DA_wireless, rxdata, wireless_dataout);

input clk, reset_sync, sample_wireless;
input[7:0] rxdata;
output cs_bar_DA_wireless;
output[7:0] wireless_dataout;

reg[2:0] state, nextstate;
reg cs_bar_DA_wireless, cs_bar_DA_wireless_int, LE_DAC, LE_DAC_int;
wire[7:0] data;

assign wireless_dataout = LE_DAC ? rxdata : 8'hz;

parameter IDLE = 0;
parameter DAC0 = 1;
parameter DAC1 = 2;
parameter DAC2 = 3;
parameter DAC3 = 4;
parameter WAIT = 5;

always @ (posedge clk)
begin
    cs_bar_DA_wireless <= cs_bar_DA_wireless_int;
    LE_DAC <= LE_DAC_int;

    if (!reset_sync) state <= IDLE;
    else state <= nextstate;
end

always @ (state or sample_wireless)
begin
    cs_bar_DA_wireless_int = 1;
    LE_DAC_int = 0;

    case (state)
      IDLE:    begin
                   nextstate = WAIT;
                 end

        WAIT:    begin
                     if (sample_wireless) nextstate = DAC0;
                     else nextstate = WAIT;
                 end

        DAC0:    begin
                     LE_DAC_int = 1;
                     nextstate = DAC1;
                 end

        DAC1:    begin
                     LE_DAC_int = 1;
                     cs_bar_DA_wireless_int = 0;
                     nextstate = DAC2;
                 end

        DAC2:    begin
                     LE_DAC_int = 1;
                     nextstate = DAC3;
                 end

        DAC3:    begin
                     LE_DAC_int = 0;
                     nextstate = WAIT;
                 end

        default: nextstate = IDLE;
    endcase
end

endmodule
```

```
// TRANSMITTER MODULE of Wireless Musical EKG

// Test Data is incrementing in a loop 00-FF
// Transmitting the Test Data at 80ms period

// Notes for myself
// RF Frequency of 915 MHz
// Included pathways.. U:\Keil\C51\INC; U:\Keil\C51\LIB
// I included the testdata used to send over a package of
// three values.

#include <chipcon/reg1010.h>
#include <chipcon/cc1010eb.h>
#include <chipcon/hal.h>
#include <stdio.h>

// Protocol constants
#define PREAMBLE_BYTE_COUNT 7

// Test packet
#define TEST_STRING_LENGTH 1
byte test_data[TEST_STRING_LENGTH];

// Initiate halTimer
byte halTimer1Usage;

//byte adc_data[TEST_STRING_LENGTH];

void main(void) {

#ifdef FREQ915

// X-tal frequency: 14.745600 MHz
// RF frequency A: 915.027455 MHz       Rx
// RF frequency B: 915.027455 MHz       Tx
// RX Mode: Low side LO
// Frequency separation: 64 kHz
// Data rate: 2.4 kBaud
// Data Format: Manchester
// RF output power: 4 dBm
// IF/RSSI: RSSI Enabled

RF_RXTXPAIR_SETTINGS code RF_SETTINGS = {
    0x4B, 0x2F, 0x15,    // Modem 0, 1 and 2
    0xAA, 0x80, 0x00,    // Freq A
    0x5C, 0xF4, 0x02,    // Freq B
    0x01, 0xAB,          // FSEP 1 and 0
    0x58,                // PLL_RX
    0x30,                // PLL_TX
    0x6C,                // CURRENT_RX
    0xF3,                // CURRENT_TX
    0x32,                // FREND
    0xFF,                // PA_POW
    0x00,                // MATCH
    0x00,                // PRESCALER
    };


#endif

#ifdef FREQ868
// X-tal frequency: 14.745600 MHz
// RF frequency A: 868.277200 MHz       Rx
// RF frequency B: 868.277200 MHz       Tx
// RX Mode: Low side LO
// Frequency separation: 64 kHz
// Data rate: 2.4 kBaud
// Data Format: Manchester
// RF output power: 4 dBm
// IF/RSSI: RSSI Enabled

RF_RXTXPAIR_SETTINGS code RF_SETTINGS = {
    0x4B, 0x2F, 0x15,    // Modem 0, 1 and 2: Manchester, 2.4 kBaud
    //0x43, 0x2F, 0x15,    // Modem 0, 1 and 2: NRZ, 2.4 kBaud
    //0xA1, 0x2F, 0x29,    // Modem 0, 1 and 2: NRZ, 38.4 kBaud
    //0xA0, 0x2F, 0x52,    // Modem 0, 1 and 2: NRZ, 76.8 kBaud
    0x75, 0xA0, 0x00,    // Freq A
    0x58, 0x32, 0x8D,    // Freq B
    0x01, 0xAB,          // FSEP 1 and 0
    0x40,                // PLL_RX
    0x30,                // PLL_TX
    0x6C,                // CURRENT_RX
    0xF3,                // CURRENT_TX
    0x32,                // FREND
    0xFF,                // PA_POW
    0x00,                // MATCH
    0x00,                // PRESCALER
    };

#endif

#ifdef FREQ433

// X-tal frequency: 14.745600 MHz
// RF frequency A: 433.302000 MHz       Rx
// RF frequency B: 433.302000 MHz       Tx
// RX Mode: Low side LO
// Frequency separation: 64 kHz
// Data rate: 2.4 kBaud
// Data Format: Manchester
// RF output power: 10 dBm
// IF/RSSI: RSSI Enabled

RF_RXTXPAIR_SETTINGS code RF_SETTINGS = {
    0x4B, 0x2F, 0x0E,    // Modem 0, 1 and 2
    0x58, 0x00, 0x00,    // Freq A
    0x41, 0xFC, 0x9C,    // Freq B
    0x02, 0x80,          // FSEP 1 and 0
    0x60,                // PLL_RX
    0x48,                // PLL_TX
    0x44,                // CURRENT_RX
    0x81,                // CURRENT_TX
    0x0A,                // FREND
    0xFF,                // PA_POW
    0xC0,                // MATCH
    0x00,                // PRESCALER
    };

#endif

        // Calibration data
    RF_RXTXPAIR_CALDATA xdata RF_CALDATA;

        // Disable watchdog timer
        WDT_ENABLE(FALSE);

    // Set optimum settings for speed and low power consumption
    MEM_NO_WAIT_STATES();
    FLASH_SET_POWER_MODE(FLASH_STANDBY_BETWEEN_READS);
```

```
        // Calibrate
    halRFCalib(&RF_SETTINGS, &RF_CALDATA);

    // Turn on RF for TX, send packet
    halRFSetRxTxOff(RF_TX, &RF_SETTINGS, &RF_CALDATA);

        // Setup UART0 with polled I/O
        UART0_SETUP(57600, CC1010EB_CLKFREQ, UART_NO_PARITY | UART_RX_TX | UART_POLLED);

        WDT_ENABLE(FALSE);
        RLED_OE(TRUE);
    YLED_OE(TRUE);
    GLED_OE(TRUE);
    BLED_OE(TRUE);

        RLED = LED_OFF;
    YLED = LED_OFF;
    GLED = LED_ON;
    BLED = LED_OFF;


        //FOR TRANSMITTER

        // Setup ADC, turn it on
                //halConfigADC(ADC_MODE_SINGLE | ADC_REFERENCE_VDD, CC1010EB_CLKFREQ, 0); // 0: Threshold value (not used in this program)

                //ADC_POWER(TRUE); // Power up ADC from sleep mode

        test_data[0] = 0;
        while (TRUE)
        {
                YLED=~YLED;
                /*
                ADC_SELECT_INPUT(ADC_INPUT_AD0);
                adc_data[0] = ADC_GET_SAMPLE_8BIT();
                ADC_SAMPLE_SINGLE();
                printf("building %03X from ADC\n",adc_data[0]);

                halRFSendPacket(PREAMBLE_BYTE_COUNT, &adc_data[0], 1);
*/

                test_data[0]++;
                //test_data[1] = 255;
                //test_data[2] = 128;

                halRFSendPacket(PREAMBLE_BYTE_COUNT, &test_data[0], 1);

                //printf("data_0 %X \n",test_data[0]);
                //printf("data_1 %X \n",test_data[1]);
                //printf("data_2 %X \n",test_data[2]);

        }
}


void halConfigADC(byte options, word clkFreq, byte threshold) {
    clkFreq= (clkFreq-250*8)/(250*16);
    ADCON2=(options&0x80)|((byte)clkFreq&0x3F);
    if (options&ADC_INTERRUPT_ENABLE) {
        EXIF&=~0x40;
        ADIE=1;
    }
    ADCON=0x80|(options&0x38);
    ADTRH=threshold;
```

```
// RECEIVER MODULE for Wireless Musical EKG

#include <chipcon/reg1010.h>
#include <chipcon/cc1010eb.h>
#include <chipcon/hal.h>
#include <stdio.h>

// Sending packages with one test string (8-bit extracted
// value from adc)
#define  RF_RX_BUF_SIZE       50
#define  TEST_STRING_LENGTH   1

//Define ASCII codes / terminal commands
#define ASCII_LF   0x0A
#define ASCII_CR   0x0D
#define ASCII_NUL  0x00
#define ASCII_ESC  0x1B

// Define RSSI limit for response
#define RSSI_LIM  -75

// Initiated Timer of Hal from Lib
byte halTimer1Usage;

// Define monitor timeouts
#define RSSI_MONITOR_TIMEOUT  0x0110
#define MAIN_MONITOR_TIMEOUT  0x00FF

void RFSetupReceive(void);

int main_monitor = 0;
int rssi_monitor = 0;
char rssi_val;
int i = 0;

byte rf_rx_string[TEST_STRING_LENGTH];
byte rf_rx_buf[RF_RX_BUF_SIZE];
byte rf_rx_index = 0;

byte rf_rx_display = FALSE;
int packet_error_cnt = 0;
byte packet_error = FALSE;


void main() {

#ifdef FREQ868

// X-tal frequency: 14.745600 MHz
// RF frequency A: 868.277200 MHz       Rx
// RF frequency B: 868.277200 MHz       Tx
// RX Mode: Low side LO
// Frequency separation: 64 kHz
// Data rate: 2.4 kBaud
// Data Format: Manchester
// RF output power: 4 dBm
// IF/RSSI: RSSI Enabled

RF_RXTXPAIR_SETTINGS code RF_SETTINGS = {
    0x4B, 0x2F, 0x15,    // Modem 0, 1 and 2: Manchester, 2.4 kBaud
    //0x43, 0x2F, 0x15,    // Modem 0, 1 and 2: NRZ, 2.4 kBaud
    //0xA1, 0x2F, 0x29,    // Modem 0, 1 and 2: NRZ, 38.4 kBaud
    //0xA0, 0x2F, 0x52,    // Modem 0, 1 and 2: NRZ, 76.8 kBaud
    0x75, 0xA0, 0x00,    // Freq A
    0x58, 0x32, 0x8D,    // Freq B
    0x01, 0xAB,          // FSEP 1 and 0
    0x40,                // PLL_RX
    0x30,                // PLL_TX
    0x6C,                // CURRENT_RX
    0xF3,                // CURRENT_TX
    0x32,                // FREND
    0xFF,                // PA_POW
    0x00,                // MATCH
    0x00,                // PRESCALER
    };

#endif

#ifdef FREQ915

// X-tal frequency: 14.745600 MHz
// RF frequency A: 915.027455 MHz       Rx
// RF frequency B: 915.027455 MHz       Tx
// RX Mode: Low side LO
// Frequency separation: 64 kHz
// Data rate: 2.4 kBaud
// Data Format: Manchester
// RF output power: 4 dBm
// IF/RSSI: RSSI Enabled

RF_RXTXPAIR_SETTINGS code RF_SETTINGS = {
    0x4B, 0x2F, 0x15,    // Modem 0, 1 and 2
    0xAA, 0x80, 0x00,    // Freq A
    0x5C, 0xF4, 0x02,    // Freq B
    0x01, 0xAB,          // FSEP 1 and 0
    0x58,                // PLL_RX
    0x30,                // PLL_TX
    0x6C,                // CURRENT_RX
    0xF3,                // CURRENT_TX
    0x32,                // FREND
    0xFF,                // PA_POW
    0x00,                // MATCH
    0x00,                // PRESCALER
    };

#endif

#ifdef FREQ433

// X-tal frequency: 14.745600 MHz
// RF frequency A: 433.302000 MHz       Rx
// RF frequency B: 433.302000 MHz       Tx
// RX Mode: Low side LO
// Frequency separation: 64 kHz
// Data rate: 2.4 kBaud
// Data Format: Manchester
// RF output power: 10 dBm
// IF/RSSI: RSSI Enabled

RF_RXTXPAIR_SETTINGS code RF_SETTINGS = {
    0x4B, 0x2F, 0x0E,    // Modem 0, 1 and 2
    0x58, 0x00, 0x00,    // Freq A
    0x41, 0xFC, 0x9C,    // Freq B
    0x02, 0x80,          // FSEP 1 and 0
    0x60,                // PLL_RX
    0x48,                // PLL_TX
    0x44,                // CURRENT_RX
    0x81,                // CURRENT_TX
    0x0A,                // FREND
    0xFF,                // PA_POW
    0xC0,                // MATCH
    0x00,                // PRESCALER
    };

#endif


    // Calibration data
    RF_RXTXPAIR_CALDATA xdata RF_CALDATA;

    // Fill up reference string
    for(i = 0; i < TEST_STRING_LENGTH; i++){
        rf_rx_string[i]=i;
    }

    // Disable watchdog timer
    WDT_ENABLE(FALSE);

    // Set optimum settings for speed and low power consumption
    MEM_NO_WAIT_STATES();
    FLASH_SET_POWER_MODE(FLASH_STANDBY_BETWEEN_READS);

    // Calibrate
    halRFCalib(&RF_SETTINGS, &RF_CALDATA);

                // Change Baud Rate
        //halRFOverrideBaudRate(RF_1200B);

    // Turn on RF for RX
    halRFSetRxTxOff(RF_RX, &RF_SETTINGS, &RF_CALDATA);

    // Setup UART0 with polled I/O
    UART0_SETUP(57600, CC1010EB_CLKFREQ, UART_NO_PARITY | UART_RX_TX | UART_POLLED);

    // Enable the LEDs
    RLED_OE(TRUE);
    YLED_OE(TRUE);
    GLED_OE(TRUE);
    BLED_OE(TRUE);

    RLED = LED_OFF;
    YLED = LED_OFF;
    GLED = LED_OFF;
    BLED = LED_OFF;

    // Setup RF receive
    RFSetupReceive();

    // Reset main loop monitor
    main_monitor = MAIN_MONITOR_TIMEOUT;

    // Reset RSSI monitor
    rssi_monitor = RSSI_MONITOR_TIMEOUT;

    // Clear screen
   /////printf("%c[2J", ASCII_ESC);

    // Place cursor at upper left corner
   /////printf("%c[H", ASCII_ESC);

    // Display RF packet data + RSSI level:
    while (TRUE) {

        // Detect RF packet error and display RF packet data:
        if(rf_rx_display == TRUE){
            rf_rx_display = FALSE;
            for(i = 2; i < TEST_STRING_LENGTH+2; i++){
                if(rf_rx_buf[i] != i-2){
                    packet_error = TRUE;
                    RLED = LED_ON;
                }
               /////printf("RXD%d=%X ", i-2, (int)rf_rx_buf[i]);
                           putchar (rf_rx_buf[i]);
            }

            if(packet_error == TRUE){
                packet_error_cnt++;
            }
            packet_error = FALSE;

           /////printf("\n");
        }else{
        }

        // Indicate main loop is running ok:
        if (main_monitor-- < 0x0000) {
            BLED = !BLED;
            main_monitor = MAIN_MONITOR_TIMEOUT;
        }

    }


    // Power down ADC:
    // In this application example this instruction does not have
    // any effect, since it is placed outside the infinite loop above.
    // However, if the "real" application needs to, for instance,
    // enter a power save mode, then it might be important power down
    // the ADC to save as much power as possible.
    ADC_POWER(FALSE);
}



// RF interrupt service rotine:
void RF_ISR (void) interrupt INUM_RF {

    INT_ENABLE(INUM_RF, INT_OFF);
        INT_SETFLAG (INUM_RF, INT_CLR);

    // Get RF receive data
    rf_rx_buf[rf_rx_index] = RF_RECEIVE_BYTE();

    RLED = LED_OFF;

    // Verify packet contents:
    switch(rf_rx_index){
        case 0:
            RF_LOCK_AVERAGE_FILTER(TRUE);
            if(rf_rx_buf[rf_rx_index] != RF_SUITABLE_SYNC_BYTE){
                RLED = LED_ON;
                        //      YLED = LED_ON;
            }else{
            }
            rf_rx_index++;
            break;

        case 1:
            if(rf_rx_buf[rf_rx_index] != TEST_STRING_LENGTH){
                RLED = LED_ON;
            }else{
            }
            rf_rx_index++;
            break;

        case TEST_STRING_LENGTH+3:
            rf_rx_index = 0;
            rf_rx_display = TRUE;
                //      YLED = LED_ON;
            PDET &= ~0x80;
            PDET |= 0x80;
            break;

        default:
            if((rf_rx_buf[rf_rx_index] != rf_rx_string[rf_rx_index-2]) && (rf_rx_index < TEST_STRING_LENGTH+2)){
                RLED = LED_ON;
            }else{
            }
            rf_rx_index++;
            break;
    }

    // Indicate packet reception
    GLED = !GLED;

    INT_ENABLE(INUM_RF, INT_ON);

        return;
}


// Setup RF for RX
void RFSetupReceive (void) {

    // Disable global interrupt
    INT_GLOBAL_ENABLE (INT_OFF);

    // Setup RF interrupt
    INT_SETFLAG (INUM_RF, INT_CLR);
    INT_PRIORITY (INUM_RF, INT_HIGH);
    INT_ENABLE (INUM_RF, INT_ON);

    // Enable RF interrupt based on bytemode
    RF_SET_BYTEMODE();

    // Setup preamble configuration
    RF_SET_PREAMBLE_COUNT(16);
    RF_SET_SYNC_BYTE(RF_SUITABLE_SYNC_BYTE);

    // Make sure avg filter is free-running + 22 baud settling time
    MODEM1=(MODEM1&0x03)|0x24;

    // Reset preamble detection
    PDET &= ~0x80;
    PDET |= 0x80;

    // Start RX
    RF_START_RX();

    // Enable global interrupt
    INT_GLOBAL_ENABLE (INT_ON);

}

/*
extern byte xdata   hal_dataformat_override,
                    hal_baudrate_override,
                    hal_pa_pow_override,
                    hal_modem0_original,
                    hal_pa_pow_original;

*/
//void halRFOverrideBaudRate(byte baudRate) {
//    if (hal_baudrate_override=baudRate)
//        MODEM0=(MODEM0&0x1F)|(baudRate&0xE0/*0x70*/    );
//              //Torgeir


//    else
//        MODEM0=(MODEM0&0x1F)|(hal_modem0_original&0xE0/*0x70*/    );
//Torgeir
//}
```

```
// TRANSMITTER MODULE of Wireless Musical EKG

// Included ADC on the EB (Evaluation Board)
// test_data is in a loop, incrementing by one 0x00-0xFF
// Increments at 12.5Hz (80ms period)

// RF Frequency of 915 MHz
// Included pathways:  U:\Keil\C51\INC; U:\Keil\C51\LIB

#include <chipcon/reg1010.h>
#include <chipcon/cc1010eb.h>
#include <chipcon/hal.h>
#include <stdio.h>

// Protocol constants
#define PREAMBLE_BYTE_COUNT 7

// Test packet
#define TEST_STRING_LENGTH 1

// Initiate halTimer
byte halTimer1Usage;
byte adc_data[TEST_STRING_LENGTH];

void main(void) {
#ifdef FREQ915

// X-tal frequency: 14.745600 MHz
// RF frequency A: 915.027455 MHz       Rx
// RF frequency B: 915.027455 MHz       Tx
// RX Mode: Low side LO
// Frequency separation: 64 kHz
// Data rate: 2.4 kBaud
// Data Format: Manchester
// RF output power: 4 dBm
// IF/RSSI: RSSI Enabled

RF_RXTXPAIR_SETTINGS code RF_SETTINGS = {
    0x4B, 0x2F, 0x15,    // Modem 0, 1 and 2
    0xAA, 0x80, 0x00,    // Freq A
    0x5C, 0xF4, 0x02,    // Freq B
    0x01, 0xAB,          // FSEP 1 and 0
    0x58,                // PLL_RX
    0x30,                // PLL_TX
    0x6C,                // CURRENT_RX
    0xF3,                // CURRENT_TX
    0x32,                // FREND
    0xFF,                // PA_POW
    0x00,                // MATCH
    0x00,                // PRESCALER
    };


#endif

#ifdef FREQ868
// X-tal frequency: 14.745600 MHz
// RF frequency A: 868.277200 MHz       Rx
// RF frequency B: 868.277200 MHz       Tx
// RX Mode: Low side LO
// Frequency separation: 64 kHz
// Data rate: 2.4 kBaud
// Data Format: Manchester
// RF output power: 4 dBm
// IF/RSSI: RSSI Enabled

RF_RXTXPAIR_SETTINGS code RF_SETTINGS = {
    0x4B, 0x2F, 0x15,    // Modem 0, 1 and 2: Manchester, 2.4 kBaud
    //0x43, 0x2F, 0x15,    // Modem 0, 1 and 2: NRZ, 2.4 kBaud
    //0xA1, 0x2F, 0x29,    // Modem 0, 1 and 2: NRZ, 38.4 kBaud
    //0xA0, 0x2F, 0x52,    // Modem 0, 1 and 2: NRZ, 76.8 kBaud
    0x75, 0xA0, 0x00,    // Freq A
    0x58, 0x32, 0x8D,    // Freq B
    0x01, 0xAB,          // FSEP 1 and 0
    0x40,                // PLL_RX
    0x30,                // PLL_TX
    0x6C,                // CURRENT_RX
    0xF3,                // CURRENT_TX
    0x32,                // FREND
    0xFF,                // PA_POW
    0x00,                // MATCH
    0x00,                // PRESCALER
    };

#endif

#ifdef FREQ433

// X-tal frequency: 14.745600 MHz
// RF frequency A: 433.302000 MHz       Rx
// RF frequency B: 433.302000 MHz       Tx
// RX Mode: Low side LO
// Frequency separation: 64 kHz
// Data rate: 2.4 kBaud
// Data Format: Manchester
// RF output power: 10 dBm
// IF/RSSI: RSSI Enabled

RF_RXTXPAIR_SETTINGS code RF_SETTINGS = {
    0x4B, 0x2F, 0x0E,    // Modem 0, 1 and 2
    0x58, 0x00, 0x00,    // Freq A
    0x41, 0xFC, 0x9C,    // Freq B
    0x02, 0x80,          // FSEP 1 and 0
    0x60,                // PLL_RX
    0x48,                // PLL_TX
    0x44,                // CURRENT_RX
    0x81,                // CURRENT_TX
    0x0A,                // FREND
    0xFF,                // PA_POW
    0xC0,                // MATCH
    0x00,                // PRESCALER
    };

#endif

        // Calibration data
    RF_RXTXPAIR_CALDATA xdata RF_CALDATA;

        // Disable watchdog timer
        WDT_ENABLE(FALSE);

    // Set optimum settings for speed and low power consumption
    MEM_NO_WAIT_STATES();
    FLASH_SET_POWER_MODE(FLASH_STANDBY_BETWEEN_READS);

        // Calibrate
    halRFCalib(&RF_SETTINGS, &RF_CALDATA);

        // Change Baud Rate
        //halRFOverrideBaudRate(RF_19200B);

    // Turn on RF for TX, send packet
    halRFSetRxTxOff(RF_TX, &RF_SETTINGS, &RF_CALDATA);


        // Setup UART0 with polled I/O
        UART0_SETUP(57600, CC1010EB_CLKFREQ, UART_NO_PARITY | UART_RX_TX | UART_POLLED);



WDT_ENABLE(FALSE);
        RLED_OE(TRUE);
    YLED_OE(TRUE);
    GLED_OE(TRUE);
    BLED_OE(TRUE);

        RLED = LED_OFF;
    YLED = LED_OFF;
    GLED = LED_ON;
    BLED = LED_OFF;


        //FOR TRANSMITTER

         //Setup ADC, turn it on
                halConfigADC(ADC_MODE_SINGLE | ADC_REFERENCE_VDD, CC1010EB_CLKFREQ, 0); // 0: Threshold value (not used in this program)

                ADC_POWER(TRUE); // Power up ADC from sleep mode

//      test_data[0] = 0;
        while (TRUE)
        {
                YLED=~YLED;

                ADC_SELECT_INPUT(ADC_INPUT_AD0);
                adc_data[0] = ADC_GET_SAMPLE_8BIT();
                ADC_SAMPLE_SINGLE();
                //printf("building %03X from ADC\n",adc_data[0]);

                halRFSendPacket(PREAMBLE_BYTE_COUNT, &adc_data[0], 1);


                //test_data[0]++;
                //test_data[1] = 255;
                //test_data[2] = 128;

        //      halRFSendPacket(PREAMBLE_BYTE_COUNT, &test_data[0], 1);

        //      printf("data_0 %X \n",test_data[0]);
                //printf("data_1 %X \n",test_data[1]);
                //printf("data_2 %X \n",test_data[2]);

        }
}


void halConfigADC(byte options, word clkFreq, byte threshold) {
    clkFreq= (clkFreq-250*8)/(250*16);
    ADCON2=(options&0x80)|((byte)clkFreq&0x3F);
    if (options&ADC_INTERRUPT_ENABLE) {
        EXIF&=~0x40;
        ADIE=1;
    }
    ADCON=0x80|(options&0x38);
    ADTRH=threshold;
}

/*
extern byte xdata   hal_dataformat_override,
                    hal_baudrate_override,
                    hal_pa_pow_override,
                    hal_modem0_original,
                    hal_pa_pow_original;
*/
/*
void halRFOverrideBaudRate(byte baudRate) {
    if (hal_baudrate_override=baudRate)
//        MODEM0=(MODEM0&0x1F)|(baudRate&0xE0/*0x70*/    //);
                //Torgeir

                /*
    else
//        MODEM0=(MODEM0&0x1F)|(hal_modem0_original&0xE0/*0x70*/    //)//;
                //Torgeir
//}
```