Appendix: Code (Group 17)

Wireless/Sensor Module

Wireless Transmission C Code

```
#include <chipcon/hal.h>
#include <chipcon/cc1010eb.h>

#define N_PREAMBLE_BYTES 3
#define PACKET_LENGTH 3

byte packet[PACKET_LENGTH];

byte selectb = 0x00;
byte count = 0xFF;

void main(void) {

    // here's our setup for RF

    // X-tal frequency: 14.745600 MHz
    // RF frequency A: 868.277200 MHz   Tx
    // RF frequency B: 868.277200 MHz   Tx
    // RX Mode: Low side LO
    // Frequency separation: 64 kHz
    // Data rate: 76.8 kBaud
    // Data Format: Manchester
    // RF output power: 0 dBm
    // IF/RSSI: RSSI Enabled

    RF_RXTXPAIR_SETTINGS code RF_SETTINGS = {
        0xA8, 0x2F, 0x52,    // Modem 0, 1 and 2
        0x58, 0x32, 0x8D,    // 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
        0xA0,                // PA_POW
        0x00,                // MATCH
        0x00,                // PRESCALER
        };

    RF_RXTXPAIR_CALDATA xdata RF_CALDATA;

    // turn off wdt
    WDT_ENABLE(FALSE);
```

```
// turn on the green and red LEDS
RLED_OE(TRUE);
GLED_OE(TRUE);

// the following is apparently optimal for speed and low power
  //  consumption
MEM_NO_WAIT_STATES();
FLASH_SET_POWER_MODE(FLASH_STANDBY_BETWEEN_READS);

// configure the ADC to read single samples with vdd as ref.
  //  voltage
halConfigADC(ADC_MODE_SINGLE | ADC_REFERENCE_VDD,
  CC1010EB_CLKFREQ, 0);

// configure the ADC to read from AD0
ADC_SELECT_INPUT(ADC_INPUT_AD0);

// turn on the ADC
ADC_POWER(TRUE);

// set P1.{0-6} to act as output pins
P1DIR = P1DIR & ~0x7F;

// set P2.0 to act as an output pin
//P2DIR = P2DIR & ~0x01;

// calibrate RF
halRFCalib(&RF_SETTINGS, &RF_CALDATA);

// turn on RF for TX
halRFSetRxTxOff(RF_TX, &RF_SETTINGS, &RF_CALDATA);


while (TRUE) {

       if (selectb == 0) {
           packet[1] = 0x0;
           P1_0 = FALSE;
           ADC_SELECT_INPUT(ADC_INPUT_AD0);
       } else if (selectb == 1) {
           packet[1] = 0x1;
           P1_0 = TRUE;
           ADC_SELECT_INPUT(ADC_INPUT_AD0);
       } else if (selectb == 2) {
           packet[1] = 0x2;
           P1_1 = FALSE;
           ADC_SELECT_INPUT(ADC_INPUT_AD1);
       } else if (selectb == 3) {
           packet[1] = 0x3;
           P1_1 = TRUE;
           ADC_SELECT_INPUT(ADC_INPUT_AD1);
       } else if (selectb == 4) {
           packet[1] = 0x4;
           ADC_SELECT_INPUT(ADC_INPUT_AD2);
       }

       // get a byte from the ADC
```

```
            ADC_SAMPLE_SINGLE();
            packet[0] = ADC_GET_SAMPLE_8BIT();

            // send the packet
            halRFSendPacket(N_PREAMBLE_BYTES, packet, PACKET_LENGTH);

            if (selectb >= 4) selectb = 0;
            else selectb += 1;

            // wait
            while (count) count--;
            count = 0xFF;
            while (count) count--;
            count = 0xFF;
            while (count) count--;
            count = 0xFF;
            while (count) count--;
            count = 0xFF;
    }
}


// stolen from halRFTest in EXAMPLES/Chipcon/RF/
// waits for at least timeOut ms
void halWait(byte timeOut, word clkFreq) {
    ulong wait;

    wait=(ulong)timeOut*(ulong)clkFreq/192;
    while (wait--);
}
```

Wireless Reception C Code

```
#include <chipcon/hal.h>
#include <chipcon/cc1010eb.h>

#define N_PREAMBLE_BYTES 3
#define PACKET_LENGTH 3

byte packet[PACKET_LENGTH];
byte count;


void main(void) {

    // 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: 76.8 kBaud
    // Data Format: Manchester
    // RF output power: 0 dBm
    // IF/RSSI: RSSI Enabled
```

```
RF_RXTXPAIR_SETTINGS code RF_SETTINGS = {
    0xA8, 0x2F, 0x52,    // Modem 0, 1 and 2
    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
    0xA0,                // PA_POW
    0x00,                // MATCH
    0x00,                // PRESCALER
    };


RF_RXTXPAIR_CALDATA xdata RF_CALDATA;

// turn off wdt
WDT_ENABLE(FALSE);

// set P1.{0-6} to output
P1DIR = P1DIR & ~0x7F;

// set P2.{0-4} to output
P2DIR = P2DIR & ~0x1F;

  // the following is apparently optimal for speed and low power
  //  consumption
MEM_NO_WAIT_STATES();
FLASH_SET_POWER_MODE(FLASH_STANDBY_BETWEEN_READS);

// calibrate RF
halRFCalib(&RF_SETTINGS, &RF_CALDATA);

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


while (TRUE) {

    // zero out the packet
    packet[0] = 0x00;
    packet[1] = 0x00;

    // receive packet
    halRFReceivePacket(10, packet, PACKET_LENGTH, NULL,
         CC1010EB_CLKFREQ);

         // write packet[0] to {P2.0 P1.6 P1.5 P1.4 P1.3 P1.2 P1.1
         //  P1.0}
    if (packet[0] & 0x00) P1_0 = TRUE;
    else P1_0 = FALSE;
    if ((packet[0] >> 0x01) & 0x01) P1_1 = TRUE;
    else P1_1 = FALSE;
    if ((packet[0] >> 0x02) & 0x01) P1_2 = TRUE;
```

```
        else P1_2 = FALSE;
        if ((packet[0] >> 0x03) & 0x01) P1_3 = TRUE;
        else P1_3 = FALSE;
        if ((packet[0] >> 0x04) & 0x01) P1_4 = TRUE;
        else P1_4 = FALSE;
        if ((packet[0] >> 0x05) & 0x01) P1_5 = TRUE;
        else P1_5 = FALSE;
        if ((packet[0] >> 0x06) & 0x01) P1_6 = TRUE;
        else P1_6 = FALSE;
        if ((packet[0] >> 0x07) & 0x01) P2_0 = TRUE;
        else P2_0 = FALSE;

        // write the three LSBs of packet[1] to P2.{3-1}
        if (packet[1] & 0x01) P2_1 = TRUE;
        else P2_1 = FALSE;
        if ((packet[1] >> 0x01) & 0x01) P2_2 = TRUE;
        else P2_2 = FALSE;
        if ((packet[1] >> 0x02) & 0x01) P2_3 = TRUE;
        else P2_3 = FALSE;

        // pulse P2.4
        count = 0xFF;
        while (count) count--;
        P2_4 = TRUE;
        count = 0xFF;
        while (count) count--;
        P2_4 = FALSE;


    }
}
```

Signal Processing Module

AlphaDisplay.v

```
//////////////////////////////////////////////////////////////////////
////////
//
// 6.111 FPGA Labkit -- Alphanumeric Display Interface
//
//
// Created: November 5, 2003
// Author: Nathan Ickes
//
//////////////////////////////////////////////////////////////////////
////////
//
// Change history
//
// 2007-02-09: Fixed register-select race condition. (Thanks to Chris
//             Buenrostro for finding this bug and David Wentzloff for
//             implementing the fix.)
// 2005-05-09: Made <dots> input registered, and converted the 640-
input MUX
//             to a 640-bit shift register.
//
//////////////////////////////////////////////////////////////////////
////////

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

   input reset; // Active high
   input 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;
reg [639:0] ldots;

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; // 0 = 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_rs <= 1'b1; // Select the control register
      disp_ce_b <= 1'b1;
      dot_index <= 31;
      state <= state+1;
   end

 8'h04:
   begin
      // Setup 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_rs <= 1'b0; // Select the dot register
      disp_ce_b <= 1'b1;
      dot_index <= 639;
      ldots <= dots;
      state <= state+1;
   end

 8'h06:
   begin
      // Load the user's dot data into the dot register
      disp_ce_b <= 1'b0;
      disp_data_out <= ldots[639];
      ldots <= ldots<<1;
      if (dot_index == 0)
    state <= 5;
      else
    dot_index <= dot_index-1;
   end
```

```
       endcase

endmodule
```

Angle.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
////////////
// Create Date:    13:57:12 04/28/2007
// Module Name:    Angle
// Description:
//                 Motion module for detecting changes in direction. It
looks for movements
//                 to one of four directions. The output is the current
direction.
//
//////////////////////////////////////////////////////////////////////////////////
////////////
module Angle(clock, reset, data, data_oen, angle_state);

    input clock, reset;  // normal inputs
    input [7:0] data; // data input
    input data_oen;      //enable for data
//   output direction;
//   output spike_count_start, count_reset;

    output [1:0] angle_state;   //direction
//   output [1:0] count;

    // the four directions
     parameter FORWARD = 0;
     parameter RIGHT = 1;
     parameter BACK = 2;
     parameter LEFT = 3;

     //Parameters signify turning clockwise or counterclockwise
     parameter CW = 0;
     parameter CCW = 1;

     // the states
     parameter WAIT_FOR_SPIKE_START = 0;
     parameter WAIT_FOR_SPIKE_END_SETUP_CW = 1;
     parameter WAIT_FOR_SPIKE_END_SETUP_CCW = 2;
     parameter WAIT_FOR_SPIKE_END_CW = 3;
     parameter WAIT_FOR_SPIKE_END_CCW = 4;
     parameter CALCULATE = 5;
     reg [2:0] state, next;

     //threshold limits on median value
     parameter IDLE_MAX = 150;
     parameter IDLE_MIN = 115;

     reg [1:0] angle_state;
     // signifies the direction a person is turning
```

```
     wire direction = (state == WAIT_FOR_SPIKE_END_CW)? CW : ((state ==
WAIT_FOR_SPIKE_END_CCW)? CCW : direction);
     // signifies that a person is currently turning
     wire spike_count_start = (state == WAIT_FOR_SPIKE_START || state ==
CALCULATE)? 0 : 1;
     // resets count
     wire count_reset = (state == WAIT_FOR_SPIKE_START)? 1 : 0;
     // count keeps track of how many quarter turns a person has taken
     wire [1:0] count;

//   LengthCounter cnt(clock, count_reset, spike_count_start, count);
    //  Sub-module to count number of quarter turns
    AngleCountFSM cnt(clock, count_reset, spike_count_start, count);

     // on posedge stick next into state
     always @(posedge clock)
        begin
            if(reset)
                state <= WAIT_FOR_SPIKE_START;
            else
                state <= next;
        end

        //on end of count or reset figure out direction
        always @(negedge spike_count_start or posedge reset)
        begin
            // on reset, angle is forward
            if ( reset )
                begin
                    angle_state <= RIGHT;
                end
            //if we are calculating and going CW
            else if( state == CALCULATE && direction == CW)
                //  this could be easily done using subtracting and
addition
                // but we had bugs with that so we manually outline each
case
                // for state +- count
                case ( angle_state )
                FORWARD:
                    case ( count )
                    2'd0: angle_state <= FORWARD;
                    2'd1:  angle_state <= RIGHT;
                    2'd2: angle_state <= BACK;
                    2'd3: angle_state <= LEFT;
                    endcase
                RIGHT:
                    case ( count )
                    2'd0: angle_state <= RIGHT;
                    2'd1:  angle_state <= BACK;
                    2'd2: angle_state <= LEFT;
                    2'd3: angle_state <= FORWARD;
                    endcase
                BACK:
                    case ( count )
                    2'd0: angle_state <= BACK;
                    2'd1:  angle_state <= LEFT;
```

```
            2'd2: angle_state <= FORWARD;
            2'd3: angle_state <= RIGHT;
            endcase
        LEFT:
            case ( count )
            2'd0: angle_state <= LEFT;
            2'd1:  angle_state <= FORWARD;
            2'd2: angle_state <= RIGHT;
            2'd3: angle_state <= BACK;
            endcase
        endcase
        //if we are calculating and going CCW
    else if( state == CALCULATE && direction == CCW)
        case ( angle_state )
        FORWARD:
            case ( count )
            2'd0: angle_state <= FORWARD;
            2'd1:  angle_state <= LEFT;
            2'd2: angle_state <= BACK;
            2'd3: angle_state <= RIGHT;
            endcase
        RIGHT:
            case ( count )
            2'd0: angle_state <= RIGHT;
            2'd1:  angle_state <= FORWARD;
            2'd2: angle_state <= LEFT;
            2'd3: angle_state <= BACK;
            endcase
        BACK:
            case ( count )
            2'd0: angle_state <= BACK;
            2'd1:  angle_state <= RIGHT;
            2'd2: angle_state <= FORWARD;
            2'd3: angle_state <= LEFT;
            endcase
        LEFT:
            case ( count )
            2'd0: angle_state <= LEFT;
            2'd1:  angle_state <= BACK;
            2'd2: angle_state <= RIGHT;
            2'd3: angle_state <= FORWARD;
            endcase
        endcase
    else
        case ( angle_state )
        FORWARD:
            angle_state <= FORWARD;
        RIGHT:
            angle_state <= RIGHT;
        BACK:
            angle_state <= BACK;
        LEFT:
            angle_state <= LEFT;
        endcase
end
```

```
/**
1)Look for the values to exceed threshold.
2)Count how long the value is exceeding threshold
3)calculate how many quarter turns that corresponds to
4)adjust angle
**/
always @(state or data_oen)
    begin
        case( state )
        WAIT_FOR_SPIKE_START:
            begin
                if( data > IDLE_MAX)
                    begin
                    next = WAIT_FOR_SPIKE_END_SETUP_CW;
                    end
                else if ( data < IDLE_MIN )
                    begin
                    next = WAIT_FOR_SPIKE_END_SETUP_CCW;
                    end
                else
                    next = WAIT_FOR_SPIKE_START;
            end
        WAIT_FOR_SPIKE_END_SETUP_CW:
            begin
                next = WAIT_FOR_SPIKE_END_CW;
            end
        WAIT_FOR_SPIKE_END_SETUP_CCW:
            begin
                next = WAIT_FOR_SPIKE_END_CCW;
            end
        WAIT_FOR_SPIKE_END_CW:
            begin
                if( data > IDLE_MAX)
                    begin
                    next = WAIT_FOR_SPIKE_END_CW;
                    end
                else
                    begin
                    next = CALCULATE;
                    end
            end
        WAIT_FOR_SPIKE_END_CCW:
            begin
                if( data < IDLE_MIN)
                    begin
                    next = WAIT_FOR_SPIKE_END_CCW;
                    end
                else
                    begin
                    next = CALCULATE;
                    end
            end
        CALCULATE:
            begin
                next = WAIT_FOR_SPIKE_START;
            end
        default:
```

```
                next = WAIT_FOR_SPIKE_START;
            endcase
        end

endmodule
```

angle_count_fsm.v

```
/**
Control center for counting number of quarter turns.
This is accomplished by to counters. One counts on the normal
clock. The other counts on an enable signal produced by first counter.
**/
module AngleCountFSM(clock, reset,  start, count);

    input clock, reset, start;
    output [1:0] count;

    wire div_enable;
    wire enable_reset = reset | (~start);
    EnableCounter ecount(clock, enable_reset, div_enable);

    wire [1:0] count;
    AngleCounter acount(div_enable, reset, count);



endmodule
```

AngleCounter.v

```
/**
Simple counter that keeps counting on every posedge of clock.
This counter overflows.
**/
module AngleCounter(clock, reset, count);

    input clock, reset;
    output [1:0] count;

    reg [1:0] count;

    always @(posedge clock or posedge reset)
    begin
        if (reset == 1)
            begin
                count <= 2'd0;
            end
        else
            begin
                count <= count + 1;
            end
    end
```

```
endmodule
```

AngleTester.v

```
module AngleTester;

    reg clock, reset, angle_oen;
    reg [7:0] angle_data;

    wire [1:0] angle_state;
    Angle angle(clock, reset, angle_data, angle_oen, angle_state);

    always #5 clock = ~clock;

    initial
    begin
        clock = 0;
        reset = 0;
        angle_oen = 0;
        angle_data = 125;

        #5
        reset = 1;
        #10
        reset = 0;

        #10
        angle_data = 175;
        angle_oen = 1;

        #10
        angle_oen = 0;

        #150
        angle_data = 125;
        angle_oen = 1;

        #10
        angle_oen = 0;

        #10
        angle_data = 175;
        angle_oen = 1;

        #10
        angle_oen = 0;

        #230
        angle_data = 125;
        angle_oen = 1;

        #10
        angle_oen = 0;

        #10
```

```
        angle_data = 40;
        angle_oen = 1;

        #10
        angle_oen = 0;

        #410
        angle_data = 125;
        angle_oen = 1;

    end

endmodule
```

Arm.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
////////////
// Create Date:    13:57:12 04/28/2007
// Module Name:    Arm
// Description:
//                Motion module to detect arm movements.
//                Does so by detecting pre-determing pattern
//                of output from an acceleromater
//
//////////////////////////////////////////////////////////////////////////////////
////////////
module Arm(clock, reset, data, data_oen, action_oen);
    // normal inputs
    input clock, reset;
    input [7:0] data;
    input data_oen;
     //output when we detect movement
    output action_oen;

     // states
     parameter IDLE = 0;
     parameter ARM_MOVING_FORWARD = 1;
     parameter ARM_EXTENDED = 2;
     parameter ARM_MOVING_BACKWARD = 3;
     reg [1:0] state, next;

     //thresholds on median
     parameter IDLE_MAX = 150;
     parameter IDLE_MIN = 70;

     // action enable is set off when we are in either of these states
     wire action_oen = (state == ARM_MOVING_FORWARD) | (state ==
ARM_EXTENDED);


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

```
                begin
                    state <= IDLE;
                end
            else
                state <= next;
        end

    //FSM looks for data to follow certain pattern.
    //If data follows pattern we should step through FSM.
    //this simple system expects correct data
    always @(state or data_oen)
        begin
            case (state)
            IDLE:
                begin
                    if( data > IDLE_MAX)
                        next = ARM_MOVING_FORWARD;
                    else
                        next = IDLE;
                end
            ARM_MOVING_FORWARD:
                begin
                    if( data > IDLE_MAX)
                        next = ARM_MOVING_FORWARD;
                    else
                        next = ARM_EXTENDED;
                end
            ARM_EXTENDED:
                begin
                    if( data < IDLE_MAX)
                        next = ARM_EXTENDED;
                    else
                        next = ARM_MOVING_BACKWARD;
                end
            ARM_MOVING_BACKWARD:
                begin
                    if( data > IDLE_MAX)
                        next = ARM_MOVING_BACKWARD;
                    else
                        next = IDLE;
                end
            endcase
        end


endmodule
```

Distributor.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
////////////
// Create Date:    13:48:36 04/28/2007
// Module Name:    Distributor
//
```

```
// Description:
//         Takes in a type and data on enable, and shuffles it out to
the appropriate
//         output corresponding to the given type and sets of the
appropriate enable
//         signal.
//
//////////////////////////////////////////////////////////////////////
///////////
module Distributor(clock, reset, type, data, enable, angle_data,
angle_oen,
                      left_arm_data, left_arm_oen, right_arm_data,
right_arm_oen,
                      left_leg_data, left_leg_oen, right_leg_data,
right_leg_oen);

     input clock, reset;
    input [2:0] type;              //type of data coming in
    input [7:0] data;           //data
    input enable;                  //enable signaling to say data is
valid

    output [7:0] angle_data;    // output to angle module
    output angle_oen;           //enable for angle module

    output [7:0] left_arm_data;         //output to arm module
    output left_arm_oen;                //enable for arm module

     output [7:0] right_arm_data;          //output to arm module
    output right_arm_oen;               //enable for arm module

     output [7:0] left_leg_data;        //output to leg modules
    output left_leg_oen;                //enable for leg modules

     output [7:0] right_leg_data;          //output to leg modules
    output right_leg_oen;               //enable for leg modules


     // The TYPES
     parameter LEFT_LEG = 0;
     parameter RIGHT_LEG = 1;
     parameter GYRO = 4;
     parameter LEFT_ARM = 2;
     parameter RIGHT_ARM = 3;

     // The starting value for outputs
     parameter IDLE = 125;

     // Assignments for all the outputs
     // Each one checks to see if type is the desired one and if enable
     // if this is true then it takes in the data, else it repeats
previous value
     wire [7:0] angle_data = ((type == GYRO) && (enable == 1))? data :
((reset == 1)? IDLE: angle_data);
    wire angle_oen        = ((type == GYRO) && (enable == 1))? 1 : 0;
```

```
    wire [7:0] left_arm_data   = ((type == LEFT_ARM) && (enable == 1))?
data : ((reset == 1)? IDLE: left_arm_data);
    wire left_arm_oen          = ((type == LEFT_ARM) && (enable == 1))?
1 : 0;

     wire [7:0] right_arm_data   = ((type == RIGHT_ARM) && (enable ==
1))? data : ((reset == 1)? IDLE: right_arm_data);
    wire right_arm_oen          = ((type == RIGHT_ARM) && (enable ==
1))? 1 : 0;

     wire [7:0] left_leg_data   = ((type == LEFT_LEG) && (enable == 1))?
data : ((reset == 1)? IDLE: left_leg_data);
    wire left_leg_oen          = ((type == LEFT_LEG) && (enable == 1))?
1 : 0;

     wire [7:0] right_leg_data   = ((type == RIGHT_LEG) && (enable ==
1))? data : ((reset == 1)? IDLE: right_leg_data);
    wire right_leg_oen          = ((type == RIGHT_LEG) && (enable ==
1))? 1 : 0;

endmodule
```

Dots

```
//////////////////////////////////////////////////////////////////////
/////////
//
// 6.111 FPGA Labkit -- Number to bitmap decoder
//
//
// Author: Yun Wu, Nathan Ickes
// Date: March 8, 2006
//
// This module converts a 4-bit input to a 80-dot (2 digit) bitmap
representing
// the numbers ' 0' through '15'.
//
//////////////////////////////////////////////////////////////////////
/////////

module Dots(clk, num, dots);
   input clk;
   input [3:0] num;
   output [79:0] dots;

   reg [79:0] dots;
   always @ (posedge clk)
     case (num)
       4'd15: dots <=
{40'b00000000_01000010_01111111_01000000_00000000, // '15'
              40'b00100111_01000101_01000101_01000101_00111001};
       4'd14: dots <=
{40'b00000000_01000010_01111111_01000000_00000000, // '14'
              40'b00011000_00010100_00010010_01111111_00010000};
```

```
       4'd13: dots <=
{40'b00000000_01000010_01111111_01000000_00000000, // '13'
              40'b00100010_01000001_01001001_01001001_00110110};
       4'd12: dots <=
{40'b00000000_01000010_01111111_01000000_00000000, // '12'
              40'b01100010_01010001_01001001_01001001_01000110};
       4'd11: dots <=
{40'b00000000_01000010_01111111_01000000_00000000, // '11'
              40'b00000000_01000010_01111111_01000000_00000000};
       4'd10: dots <=
{40'b00000000_01000010_01111111_01000000_00000000, // '10'
              40'b00111110_01010001_01001001_01000101_00111110};
       4'd09: dots <=
{40'b00000000_00000000_00000000_00000000_00000000, // ' 9'
              40'b00000110_01001001_01001001_00101001_00011110};
       4'd08: dots <=
{40'b00000000_00000000_00000000_00000000_00000000, // ' 8'
              40'b00110110_01001001_01001001_01001001_00110110};
       4'd07: dots <=
{40'b00000000_00000000_00000000_00000000_00000000, // ' 7'
              40'b00000001_01110001_00001001_00000101_00000011};
       4'd06: dots <=
{40'b00000000_00000000_00000000_00000000_00000000, // ' 6'
              40'b00111100_01001010_01001001_01001001_00110000};
       4'd05: dots <=
{40'b00000000_00000000_00000000_00000000_00000000, // ' 5'
              40'b00100111_01000101_01000101_01000101_00111001};
       4'd04: dots <=
{40'b00000000_00000000_00000000_00000000_00000000, // ' 4'
              40'b00011000_00010100_00010010_01111111_00010000};
       4'd03: dots <=
{40'b00000000_00000000_00000000_00000000_00000000, // ' 3'
              40'b00100010_01000001_01001001_01001001_00110110};
       4'd02: dots <=
{40'b00000000_00000000_00000000_00000000_00000000, // ' 2'
              40'b01100010_01010001_01001001_01001001_01000110};
       4'd01: dots <=
{40'b00000000_00000000_00000000_00000000_00000000, // ' 1'
              40'b00000000_01000010_01111111_01000000_00000000};
       4'd00: dots <=
{40'b00000000_00000000_00000000_00000000_00000000, // ' 0'
              40'b00111110_01010001_01001001_01000101_00111110};
       // No default case, becase every case is already accounted for.
     endcase

endmodule
```

Duck.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////
//////////
// Create Date:    13:57:12 04/28/2007
// Module Name:    Duck
// Description:
```

```
//                Motion module to detect ducking movement.
//                Does so by detecting pre-determing pattern
//                of output from two accelerometers.
//
//
////////////////////////////////////////////////////////////////////////
///////////
module Duck(clock, reset, jump_oen, leg1_data, leg1_data_oen,
leg2_data, leg2_data_oen, action_oen);

    input clock, reset, jump_oen;   //normal inputs
     //data inputs
    input [7:0] leg1_data;
    input leg1_data_oen;
     input [7:0] leg2_data;
    input leg2_data_oen;

     //enable output
    output action_oen;

     //states
     parameter IDLE = 0;
     parameter MOVING_DOWN = 1;
     parameter DOWN = 2;
     parameter RETURNING = 3;
     reg [1:0] state, next;

     //thresholds on median
     parameter IDLE_MIN = 130;
     parameter IDLE_MAX = 180;

     //action is enabled when we are in these states
     wire action_oen = ((state == MOVING_DOWN) | (state == DOWN) |
(state == RETURNING));

     always @(posedge clock)
        begin
            if (reset)
                state <= IDLE;
            else
                state <= next;
        end

    //FSM looks for data to follow certain pattern.
    //If data follows pattern we should step through FSM.
    //this simple system expects correct data
    // we should detect a spike downwards (acceleration downwards)
    // followed by a positive spike (deceleration upwards) as we reach
bottom
    // then positive spike (acceleration up) should continue as we go
back up
    // finally when we hit ground, spike downwards (deceleration
downwards)
    always @(state or leg1_data_oen or leg2_data_oen or jump_oen)
        begin
            case (state)
            IDLE:
```

```
                begin
                    if( (leg1_data < IDLE_MIN) && (leg2_data < IDLE_MIN)
&& (jump_oen == 0))
                        next =  MOVING_DOWN;
                    else
                        next = IDLE;
                end
            MOVING_DOWN:
                begin
                    begin
                    if( (leg1_data > IDLE_MIN) && (leg2_data > IDLE_MIN)
)
                        next =  DOWN;
                    else
                        next = MOVING_DOWN;
                end
                end
            DOWN:
                begin
                    if( (leg1_data < IDLE_MIN) && (leg2_data < IDLE_MIN)
)
                        next = RETURNING;
                    else
                        next = DOWN;
                end
            RETURNING:
                begin
                    if( (leg1_data > IDLE_MIN) && (leg2_data > IDLE_MIN)
)
                        next = IDLE;
                    else
                        next = RETURNING;
                end
            endcase
        end


endmodule
```

enable_counter.v

```
/**
Simple counter. Counts to a given value. When it
reaches value, sends out an enable and restarts.
**/
module EnableCounter(clock, reset, div_enable);

    input clock, reset;
    output div_enable;

    reg div_enable;
    reg[31:0] count;

    parameter LIMIT = 32'd10750000;  //Estimated time for quarter turn
                                              //based on 27Mhz clock
```

```
    always @(posedge clock)
    begin
        if (reset == 1)
            begin
                count <= 32'd0;
                div_enable <= 1'b0;
            end
        else if (count == LIMIT)
            begin
                div_enable <= 1'b1;
                count <= 32'd0;
            end
        else
            begin
                count <= count + 1;
                div_enable <= 1'b0;
            end
    end

endmodule
```

interpreter.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
////////////
// Create Date:    14:05:32 04/28/2007
// Module Name:    Interpreter
// Description:
//
//     Interpreter takes in motion enables from motion modules. Using
those enables
//     it decides whether an action was performed. Actions consist of
//     left, up, right, down, a, and b.
//////////////////////////////////////////////////////////////////////////////////
////////////
module Interpreter(clock,
                     angle_state, left_arm_oen, right_arm_oen,
walk_oen, jump_oen, duck_oen,
                     left, up, right, down, a, b);

    input clock;  //normal input

     // motion modules outputs
    input [1:0] angle_state;
    input left_arm_oen;
    input right_arm_oen;
    input walk_oen;
    input jump_oen;
    input duck_oen;

     // action outputs
    output left;
```

```
    output up;
    output right;
    output down;
    output a;
    output b;

     // states for angle
     parameter FORWARD = 0;
     parameter RIGHT = 1;
     parameter BACK = 2;
     parameter LEFT = 3;

     reg left;
    reg up;
    reg right;
    reg down;
    reg a;
    reg b;

     // put together motion enables for specific actions
     always @(posedge clock)
        begin
            if(jump_oen)
                up = 1;
            else
                up = 0;

            if(duck_oen)
                down = 1;
            else
                down = 0;

            if(left_arm_oen)
                a = 1;
            else
                a = 0;

            if(right_arm_oen)
                b = 1;
            else
                b = 0;

            if(walk_oen == 1 && angle_state == LEFT)
                left = 1;
            else
                left = 0;

            if(walk_oen == 1 && angle_state == RIGHT)
                right = 1;
            else
                right = 0;
        end


endmodule
```

Jump.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
////////////
// Create Date:    13:57:12 04/28/2007
// Module Name:    Jump
// Description:
//                Motion module to detect jumping movement.
//                Does so by detecting pre-determing pattern
//                of output from two accelerometers.
//
//
//////////////////////////////////////////////////////////////////////////////////
////////////
module Jump(clock, reset, duck_oen, leg1_data, leg1_data_oen,
leg2_data, leg2_data_oen, action_oen);

    input clock, reset, duck_oen;   //normal inputs
     //data inputs
    input [7:0] leg1_data;
    input leg1_data_oen;
     input [7:0] leg2_data;
    input leg2_data_oen;

     //action enable
    output action_oen;

    //states
     parameter IDLE = 0;
    parameter MOVING_UP = 1;
     parameter UP = 2;
     parameter RETURNING = 3;
     reg [1:0] state, next;

     //thresholds on median
     parameter IDLE_MIN = 125;
     parameter IDLE_MAX = 180;

     //action is enabled when we are in these states
     wire action_oen = ((state == MOVING_UP) | (state == UP) | (state ==
RETURNING));

     always @(posedge clock)
        begin
            if (reset)
                state <= IDLE;
            else
                state <= next;
        end

    //FSM looks for data to follow certain pattern.
    //If data follows pattern we should step through FSM.
    //this simple system expects correct data
    // we should detect a spike upwards (acceleration upwards)
```

```
    // followed by a negative spike (deceleration downwards) as we reach
max of jump
    // then negative spike (acceleration down) should continue as we go
down
    // finally when we hit ground, spike upwards (deceleration upwards)
    always @(state or leg1_data_oen or leg2_data_oen or duck_oen)
        begin
            case (state)
            IDLE:
                begin
                    if( (leg1_data > IDLE_MAX) && (leg2_data > IDLE_MAX)
&& (duck_oen == 0))
                        next =  MOVING_UP;
                    else
                        next = IDLE;
                end
            MOVING_UP:
                begin
                    if( (leg1_data < IDLE_MAX) && (leg2_data < IDLE_MAX)
)
                        next =  UP;
                    else
                        next = MOVING_UP;
                end
            UP:
                begin
                    if( (leg1_data > IDLE_MAX) && (leg2_data > IDLE_MAX)
)
                        next = RETURNING;
                    else
                        next = UP;
                end
            RETURNING:
                begin
                    if( (leg1_data < IDLE_MAX) && (leg2_data < IDLE_MAX)
)
                        next =  IDLE;
                    else
                        next = RETURNING;
                end
            endcase
        end

endmodule
```

labkit.v

<standard labkit inputs and assigns omitted>

```
    //clock
    wire clock = clock_27mhz;

    // reset button handling
    wire reset_not, reset;
```

```
    assign reset = ~reset_not;   // invert button because active high
    debounce db_reset(0,clock, button0, reset_not);

    //data
    wire [7:0] data_u, data;
    assign data_u = user4[7:0];//switch[7:0];
    debounce8 data_db(0,clock, data_u, data);
    assign user4[27:8] = 20'hz;
    // type is entered by using left/right buttons to select
    wire [2:0] type_u, type;
    assign type_u = user4[30:28];
    debounce3 type_db(0,clock, type_u, type);
    //assign type = 3'b011;  //4 bits for dot matrix display
    //wire left_clean, left, right_clean, right;
    // synchronize buttons
    //debounce db_left(0,clock, button_left, left_clean);
    //debounce db_right(0,clock, button_right, right_clean);
    // turn button inputs to pulses rather than levels
    //LevelToPulse ltp_left(clock, reset, left_clean, left);
    //LevelToPulse ltp_right(clock, reset, right_clean, right);
    // feed to selector module
    //SelectType selector(clock, reset, left, right, type);


   // button0, button1, button2, button3, button_enter, button_right,
   // button_left, button_down, button_up, and switches are inputs

    // button to submit data
    wire enable_u, enable_level, enable;
    //assign button_not = ~button_enter;
    assign enable_u = user4[31];
    debounce_fast enable_db(0,clock, enable_u, enable);
    //LevelToPulse ltp_enable(clock, reset, enable_level, enable);
    //debounce db_submit(0, clock, button_not, enable_level);


    //distibutor takes in data and feeds it out
    wire [7:0] angle_data, left_arm_data, right_arm_data, left_leg_data,
right_leg_data;
    wire angle_oen, left_arm_oen, right_arm_oen, left_leg_oen,
right_leg_oen;
    Distributor dist(clock, reset, type, data, enable, angle_data,
angle_oen,
                     left_arm_data, left_arm_oen, right_arm_data,
right_arm_oen,
                     left_leg_data, left_leg_oen, right_leg_data,
right_leg_oen);



    ////////////////////////////////////////////
    //following is instantiations of all motion modules
    ////////////////////////////////////////////
    wire [1:0] angle_state;
    Angle angle(clock, reset, angle_data, angle_oen, angle_state);

    wire left_arm_action_oen;
```

```
    Arm leftarm(clock, reset, left_arm_data, left_arm_oen,
left_arm_action_oen);

    wire right_arm_action_oen;
    Arm rightarm(clock, reset, right_arm_data, right_arm_oen,
right_arm_action_oen);

    wire duck_action_oen;
    wire jump_action_oen;
    Duck duck(clock, reset, jump_action_oen, left_leg_data,
left_leg_oen, right_leg_data, right_leg_oen, jump_action_oen);
    Jump jump(clock, reset, duck_action_oen, left_leg_data,
left_leg_oen, right_leg_data, right_leg_oen, duck_action_oen);

    wire walk_action_oen;
    wire count_end;
    Walking walking(clock, reset, jump_action_oen, left_leg_data,
left_leg_oen, right_leg_data, right_leg_oen, walk_action_oen,
count_end);


    ////////////////////////////////////////////////
    //following is instantiations of interpreter
    ////////////////////////////////////////////////
    wire zleft, zup, zright, zdown, za, zb;
    Interpreter interpret(clock,
                        angle_state, left_arm_action_oen,
right_arm_action_oen, walk_action_oen, jump_action_oen,
duck_action_oen,
                        zleft, zup, zright, zdown, za, zb);

    assign user3[6:3] = {zdown, zleft, zup, zright};
    assign user3[1:0] = {za, zb};

    ////////////////////////////////////////////////
    // when an action for a particular type is set off led is lit
    assign led[7:0] = {~left_arm_action_oen , ~right_arm_action_oen ,
~duck_action_oen , ~jump_action_oen , ~walk_action_oen, ~count_end,
reset, enable_level};

    // display type on dot matrix display
    wire [79:0] type_dot;
    Dots dots1(clock, {0,type}, type_dot);

    // display angle on dot matrix display
    wire [79:0] dot2;
    wire [3:0] angle_dot = { 2'b00, angle_state };
    Dots dots2(clock, angle_dot, dot2);

    wire [79:0] left_dot, up_dot, right_dot, down_dot, a_dot, b_dot;
    Dots dots3(clock, {3'b000, zleft} , left_dot);
    Dots dots4(clock, {3'b000, zup} , up_dot);
    Dots dots5(clock, {3'b000, zright}, right_dot);
    Dots dots6(clock, {3'b000, zdown}, down_dot);
    Dots dots7(clock, {3'b000, za}, a_dot);
    Dots dots8(clock, {3'b000, zb}, b_dot);
```

```
    assign analyzer1_data = {left_arm_data, right_arm_data};
   assign analyzer1_clock = clock;
   assign analyzer2_data = {left_leg_data, right_leg_data};
   assign analyzer2_clock = clock;
   assign analyzer3_data = {user4[31:28], 3'hF ,enable_level, enable,
type, 4'hF} ;
   assign analyzer3_clock = clock;
   assign analyzer4_data = {angle_data, 6'hFF, angle_state };
   assign analyzer4_clock = clock;

    //make dots
    wire [639:0] dots = {dot2, type_dot, a_dot, b_dot, left_dot, up_dot,
right_dot, down_dot};
    // LED Displays
   wire disp_blank;
   wire disp_clock;
   wire disp_rs;
   wire disp_ce_b;
   wire disp_reset_b;
   wire disp_data_out;
   // handle dot display
    AlphaDisplay adis(reset, clock, disp_blank, disp_clock, disp_rs,
disp_ce_b,
       disp_reset_b, disp_data_out, dots);


endmodule
```

SelectType.v

```
module SelectType(clock, reset, left, right, type);

     input clock, reset, left, right;
     output [2:0] type;

     parameter LEFT_LEG = 0;
     parameter RIGHT_LEG = 1;
     parameter GYRO = 2;
     parameter LEFT_ARM = 3;
     parameter RIGHT_ARM = 4;
     reg [2:0] type, next;


     always @(posedge clock)
        begin
            if( reset )
                type <= LEFT_LEG;
            else
                type <= next;
        end

     always @(type or left or right)
        begin
            case ( type )
            LEFT_LEG:
```

```
                if ( left == 1 && right == 0)
                    next = RIGHT_ARM;
                else if ( left == 0 && right == 1)
                    next = RIGHT_LEG;
                else
                    next = LEFT_LEG;
            RIGHT_LEG:
                if ( left == 1 && right == 0)
                    next = LEFT_LEG;
                else if ( left == 0 && right == 1)
                    next = GYRO;
                else
                    next = RIGHT_LEG;
            GYRO:
                if ( left == 1 && right == 0)
                    next = RIGHT_LEG;
                else if ( left == 0 && right == 1)
                    next = LEFT_ARM;
                else
                    next = GYRO;
            LEFT_ARM:
                if ( left == 1 && right == 0)
                    next = GYRO;
                else if ( left == 0 && right == 1)
                    next = RIGHT_ARM;
                else
                    next = LEFT_ARM;
            RIGHT_ARM:
                if ( left == 1 && right == 0)
                    next = LEFT_ARM;
                else if ( left == 0 && right == 1)
                    next = LEFT_LEG;
                else
                    next = RIGHT_ARM;
            endcase
        end

endmodule
```

SimpleCounter.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
////////////
// Create Date:    13:57:12 04/28/2007
// Design Name:
// Module Name:    SpikeUPCounter
// Project Name:
// Description:
//
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
```

```
//////////////////////////////////////////////////////////////////////
///////////
module SimpleCounter(clock, start, count_end);

    input clock, start;
    output count_end;


    parameter END_POINT = 13500000;//half a second on 27Mhz clock

    reg [31:0] count;
    reg count_end;

    always @(posedge clock)
        begin
            if( !start )
                begin
                    count <= 0;
                    count_end <= 0;
                end
            else   if( count == END_POINT)
                       begin
                           count <= count;
                           count_end <= 1;
                       end
            else
                begin
                           count <= count + 1;
                           count_end <= 0;
                end
        end

endmodule
```

TestDistributor.v

```
module TestDistributor;

reg clock, reset;
reg [2:0] type;
reg [7:0] data;
reg enable_level;
wire enable;

LevelToPulse ltp_right(clock, reset, enable_level, enable);

wire [7:0] angle_data, left_arm_data, right_arm_data, left_leg_data,
right_leg_data;
wire angle_oen, left_arm_oen, right_arm_oen, left_leg_oen,
right_leg_oen;
Distributor dist(clock,reset, type, data, enable, angle_data,
angle_oen,
                   left_arm_data, left_arm_oen, right_arm_data,
right_arm_oen,
```

```
                        left_leg_data, left_leg_oen, right_leg_data,
right_leg_oen);

wire right_arm_action_oen;
Arm rightarm(clock, reset, right_arm_data, right_arm_oen,
right_arm_action_oen);

wire left_arm_action_oen;
Arm leftarm(clock, reset, left_arm_data, left_arm_oen,
left_arm_action_oen);

always #5 clock = ~clock;

initial
    begin
    clock = 0;
    reset = 0;
    type = 4;
    data = 8'hFF;
    #5
    reset = 1;
    #10
    reset = 0;
    #30
    enable_level = 1;
    #10
    enable_level = 0;

    #30
    data = 8'hF2;
    enable_level = 1;
    #10
    enable_level = 0;

    #30
    data = 8'h64;
    enable_level = 1;
    #10
    enable_level = 0;

    #30
    data = 8'hFF;
    type = 3;
    enable_level = 1;
    #10
    enable_level = 0;

    #20
    type = 4;
    data = 8'hD4;
    enable_level = 1;
    #10
    enable_level = 0;

    #30
    data = 8'h68;
    enable_level = 1;
```

```
    #10
    enable_level = 0;

    end

endmodule
```

TestWalker.v

```
module TestWalker;

reg clock, reset, jump_action_oen,left_leg_oen, right_leg_oen;
reg [7:0] left_leg_data, right_leg_data;

wire walk_action_oen;
Walking walking(clock, reset, jump_action_oen, left_leg_data,
left_leg_oen, right_leg_data, right_leg_oen, walk_action_oen);

always #5 clock = ~clock;
initial
    begin
    clock = 0;
    reset = 1;
    jump_action_oen = 0;
    left_leg_oen = 0;
    right_leg_oen = 0;
    left_leg_data = 125;
    right_leg_data = 125;

    #15
    reset = 0;

    #10
    left_leg_oen = 1;
    right_leg_oen = 1;

    #10
    left_leg_oen = 0;
    right_leg_oen = 0;

    #10
    right_leg_oen = 1;
    right_leg_data = 151;

    #10
    right_leg_oen = 0;

    #10
    right_leg_oen = 1;
    right_leg_data = 100;

    #10
    right_leg_oen = 0;

    #10
```

```
    right_leg_oen = 1;
    right_leg_data = 151;

    #10
    right_leg_oen = 0;

    #10
    right_leg_oen = 1;
    right_leg_data = 125;

    #10
    right_leg_oen = 0;

    end

endmodule
```

Walking.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
////////////
// Create Date:    13:57:12 04/28/2007
// Module Name:    Walking
// Description:
//
//
//
//////////////////////////////////////////////////////////////////////////////////
////////////
module Walking(clock, reset, jumping_oen, leg1_data, leg1_data_oen,
                  leg2_data, leg2_data_oen, action_oen, count_end);

     input clock, reset, jumping_oen;   //normals inputs
     //data
    input [7:0] leg1_data;
    input leg1_data_oen;
     input [7:0] leg2_data;
    input leg2_data_oen;

    output action_oen;   // action enable output
     output count_end;   //debugging output

     //states
     parameter IDLE = 0;
    parameter RIGHT_MOVING_UP = 1;
     parameter RIGHT_UP = 2;
     parameter RIGHT_RETURNING = 3;
     parameter LEFT_MOVING_UP = 4;
     parameter LEFT_UP = 5;
     parameter LEFT_RETURNING = 6;
     reg [2:0] state, next;


     //thresholds on median
```

```
     parameter IDLE_MIN = 125;
     parameter IDLE_MAX = 170;

     // sets action enable
     wire action_oen = ( (state == IDLE) || (state == RIGHT_MOVING_UP)
|| (state == LEFT_MOVING_UP) ) ?
                               ( ((count_end == 0) && (reset == 0))?
action_oen : 0 ) : 1;
    // reset for carry-on counter
     wire count_start = (state == IDLE)? ((reset == 1)? 0 : 1) : 0;
     //signifies end of count
     wire count_end;

     // initialize simple counter
     SimpleCounter sc(clock, count_start, count_end);


     always @(posedge clock)
        begin
            if( reset )
                state <= IDLE;
            else
                state <= next;
        end

    //FSM looks for data to follow certain pattern.
    //If data follows pattern we should step through FSM.
    //this simple system expects correct data
    // we should detect per leg a spike upwards (acceleration upwards)
    // followed by a negative spike (deceleration downwards) as we reach
max of leg movement
    // then negative spike (acceleration down) should continue as we go
down
    // finally when we hit ground, spike upwards (deceleration upwards)
    //     We should alternate this pattern for each leg.
    //to maintain a timer after each leg movement. If timer expires
    //walking motion is over.
    always @(state or leg1_data_oen or leg2_data_oen or count_end)
        begin
            case( state )
            IDLE:
                begin
                    if( (leg1_data > IDLE_MAX) && (leg2_data < IDLE_MAX)
&& (leg2_data > IDLE_MIN))
                        next = RIGHT_MOVING_UP;
                    else if( (leg2_data > IDLE_MAX) && (leg1_data <
IDLE_MAX) && (leg1_data > IDLE_MIN))
                        next = LEFT_MOVING_UP;
                    else
                        next = IDLE;
                end
            RIGHT_MOVING_UP:
                begin
                    if( (leg1_data < IDLE_MAX) && (leg2_data < IDLE_MAX)
&& (leg2_data > IDLE_MIN))
                        next =  RIGHT_UP;
```

```
                    else if( (leg1_data > IDLE_MAX) && (leg2_data <
IDLE_MAX) && (leg2_data > IDLE_MIN))
                        next = RIGHT_MOVING_UP;
                    else
                        next = IDLE;
                end
            RIGHT_UP:
                begin
                    if( (leg1_data > IDLE_MAX) && (leg2_data < IDLE_MAX)
&& (leg2_data > IDLE_MIN))
                        next = RIGHT_RETURNING;
                    else if( (leg1_data < IDLE_MAX) && (leg2_data <
IDLE_MAX) && (leg2_data > IDLE_MIN))
                        next = RIGHT_UP;
                    else
                        next = IDLE;
                end
            RIGHT_RETURNING:
                begin
                    if( (leg1_data < IDLE_MAX) )
                        next =  IDLE;
                    else
                        next = RIGHT_RETURNING;
                end
            LEFT_MOVING_UP:
                begin
                    if( (leg2_data < IDLE_MAX) && (leg1_data < IDLE_MAX)
&& (leg1_data > IDLE_MIN))
                        next =  LEFT_UP;
                    else if( (leg2_data > IDLE_MAX) && (leg1_data <
IDLE_MAX) && (leg1_data > IDLE_MIN))
                        next = LEFT_MOVING_UP;
                    else
                        next = IDLE;
                end
            LEFT_UP:
                begin
                    if( (leg2_data > IDLE_MAX) && (leg1_data < IDLE_MAX)
&& (leg1_data > IDLE_MIN))
                        next = LEFT_RETURNING;
                    else if( (leg2_data < IDLE_MAX) && (leg1_data <
IDLE_MAX) && (leg1_data > IDLE_MIN))
                        next = LEFT_UP;
                    else
                        next = IDLE;
                end
            LEFT_RETURNING:
                begin
                    if( (leg2_data < IDLE_MAX) )
                        next =  IDLE;
                    else
                        next = LEFT_RETURNING;
                end
            endcase
        end
```

```
endmodule
```

PS/2

ps2_break_code_gen.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
//
// Ceryen Tan
//
// Generates break code from a make code
//
//////////////////////////////////////////////////////////////////////////////////
module ps2_break_code_gen(make_code, break_code);
    input [15:0] make_code;
    output [23:0] break_code;

    assign break_code =
        (make_code[15:8] == 8'b1110_0000) ?
            {16'b1110_0000_1111_0000, make_code[7:0]} :
            {8'b1111_0000, make_code};
endmodule
```

ps2_clock_gen.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
//
// Ceryen Tan
//
// Communication over the PS/2 interface is dependent on the PS/2 clock
signal.
// The PS/2 clock signal determines how quickly data is serialized onto
the
// the data busline.  When the host wants to send data to the device,
the PS/2
// clock signal determines how quickly the host shifts bits onto the
serial
// data line.  When the device wants to send data to the host, the PS/2
clock
// signal also determines how quickly the device shifts bits onto the
serial
// data line.
//
// Note that the device is responsible for generating the PS/2 clock
signal in
```

```
// *both* host-to-device communication and device-to-host
communication.  This
// means that when the host sends data to the device, the device itself
// determines how quickly data is sent over the data line.
//
// Also note that the PS/2 clock is only active when data is being sent
or
// received.  At all other times, the PS/2 clock line is left high.  In
other
// words, the PS/2 clock is only active when the
serializer/deserializer is
// active.
//
// This module is responsible for generating the PS/2 clock.  It
generates a
// PS/2 clock signal when clock enable is held high.  At all other
times, the
// PS/2 clock signal is left high.  Clock enable is activated by either
the
// serializer or the deserialzer.
//
// The PS/2 clock must be within a frequency range of 10-16.7 kHz.
This
// module generates a frequency of 13.5 kHz.
//
// When the clock enable signal is asserted high, this clock generator
starts
// with the clock signal being high.
//
//////////////////////////////////////////////////////////////////////
//////////

module ps2_clock_gen(reset, clk_27_mhz, ps2_clock_enable,
ps2_clock_out);
    input reset;
    input clk_27_mhz;

    // Input that determines if the clock generator is active
    input ps2_clock_enable;

    // Output clock signal
    output ps2_clock_out;

    // Parameter that determines how many 27 MHz clock cycles to count
before
    // flipping the PS/2 clock signal.  With this parameter set to 1000,
a clock
    // frequency of 13.5 kHz will be generated
    parameter MAX_CLOCK_COUNT = 1000;

    // Counts the number of 27 MHz clock cycles between PS/2 clock
signal flips
    reg [9:0] clock_27_mhz_count;

    // Output clock signal
    reg ps2_clock_out;
```

```
    initial
        begin
            clock_27_mhz_count = 0;
            ps2_clock_out = 1;
        end

    always @(posedge clk_27_mhz)
        // If reset or clock generator is not enabled, set the output
PS/2 clock
        // signal high and reset the clock count
        if(reset || !ps2_clock_enable)
            begin
                ps2_clock_out <= 1;
                clock_27_mhz_count <= 0;
            end

        // If time to flip PS/2 clock signal, flip PS/2 clock signal and
reset
        // timer
        else if(clock_27_mhz_count == MAX_CLOCK_COUNT - 1)
            begin
                ps2_clock_out <= ~ps2_clock_out;
                clock_27_mhz_count <= 0;
            end

        // Count time since last flip of PS/2 clock signal
        else
            begin
                ps2_clock_out <= ps2_clock_out;
                clock_27_mhz_count <= clock_27_mhz_count + 1;
            end
endmodule
```

ps2_control_fsm.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////
////////////
//
// Ceryen Tan
//
// This module is responsible for sending the self-test completion code
on
// startup of the computer.  This module is also responsible for
responding
// to any commands received through the deserializer.  When a command
is
// received, this module tells both the keyboard buffer and the
serializer
// buffer to clear.  This module then enqueues responses to commands to
the
// keyboard buffer so that they can be sent to the computer.  The self-
test
// completion code is also sent through the keyboard buffer.
//
```

```
//////////////////////////////////////////////////////////////////////
///////////
module ps2_control_fsm(reset, clk_27_mhz,
                           keyboard_buffer_enqueue_data,
keyboard_buffer_data_to_enqueue,
                           keyboard_buffer_clear, serial_buffer_clear,
                           deserial_received_data,
deserial_data_received);
    input reset;
    input clk_27_mhz;

    // Allows enqueuing of messages to the keyboard buffer, such as the
    // self-test completion code and responses to commands
    output keyboard_buffer_enqueue_data;
    output [23:0] keyboard_buffer_data_to_enqueue;

    // Clears the keyboard and the serial buffers
    output keyboard_buffer_clear;
    output serial_buffer_clear;

    input deserial_received_data;
    input [7:0] deserial_data_received;

    // Amount of time until self-test completion code is sent
    parameter NUM_CYCLES_BEFORE_SELF_TEST = 13500000;

    // States for the control FSM
    parameter WAIT_TO_SEND_SELF_TEST = 0;      // Wait until time to
send self-test completion code
    parameter SEND_SELF_TEST = 1;                  // Enqueue self-test
completion code to keyboard buffer
    parameter IDLE = 2;                            // Wait for message
to be received
    parameter SEND_ACK = 3;                        // Send
acknowledgement to a message
    parameter SEND_DEVICE_ID = 4;                  // Send device ID
    parameter SEND_ECHO = 5;                   // Send echo

    reg [2:0] state;

    reg [23:0] clock_timer;

    // Allows enqueuing of messages to the keyboard buffer, such as the
    // self-test completion code and responses to commands
    reg keyboard_buffer_enqueue_data;
    reg [23:0] keyboard_buffer_data_to_enqueue;

    // Clears the keyboard and the serial buffers
    reg keyboard_buffer_clear;
    reg serial_buffer_clear;

    // Stores the last message received by the module
    reg [7:0] last_data_received;

    initial
        begin
            state <= WAIT_TO_SEND_SELF_TEST;
```

```
            clock_timer <= 0;

            keyboard_buffer_enqueue_data <= 0;
            keyboard_buffer_data_to_enqueue <= 0;

            keyboard_buffer_clear <= 0;
            serial_buffer_clear <= 0;

            last_data_received <= 8'b0;
        end

    always @(posedge clk_27_mhz)
        begin
            clock_timer <= 0;

            keyboard_buffer_enqueue_data <= 0;
            keyboard_buffer_data_to_enqueue <= 0;

            keyboard_buffer_clear <= 0;
            serial_buffer_clear <= 0;

            last_data_received <= last_data_received;

            if(reset)
                begin
                    state <= WAIT_TO_SEND_SELF_TEST;

                    keyboard_buffer_clear <= 1;
                    serial_buffer_clear <= 1;

                    last_data_received <= 8'b0;
                end
            else
                case(state)
                    // Wait until time to send self-test completion code
                    WAIT_TO_SEND_SELF_TEST:
                        if(clock_timer == NUM_CYCLES_BEFORE_SELF_TEST - 1)
                            begin
                                state <= SEND_SELF_TEST;
                                clock_timer <= 0;

                                keyboard_buffer_clear <= 1;
                                serial_buffer_clear <= 1;
                            end
                        else
                            begin
                                state <= WAIT_TO_SEND_SELF_TEST;
                                clock_timer <= clock_timer + 1;

                                keyboard_buffer_clear <= 1;
                                serial_buffer_clear <= 1;
                            end

                    // Enqueue self-test completion code to keyboard
buffer
                    SEND_SELF_TEST:
```

```
                        begin
                            state <= IDLE;

                            keyboard_buffer_enqueue_data <= 1;
                            keyboard_buffer_data_to_enqueue <=
24'b1010_1010_0000_0000_0000_0000;

                            keyboard_buffer_clear <= 0;
                            serial_buffer_clear <= 0;
                        end

                    // Wait for message to be received
                    IDLE:
                        if(deserial_received_data)
                            begin
                                if(deserial_data_received == 8'b1110_1110)
    // Echo request
                                    state <= SEND_ECHO;
                                else
                                    state <= SEND_ACK;

                                last_data_received <=
deserial_data_received;

                                keyboard_buffer_clear <= 1;
                                serial_buffer_clear <= 1;
                            end
                        else
                            begin
                                state <= IDLE;

                                keyboard_buffer_clear <= 0;
                                serial_buffer_clear <= 0;
                            end

                    // Send acknowledgement to a message
                    SEND_ACK:
                        begin
                            if(last_data_received == 8'b1111_1111)
        // Reset command
                                state <= SEND_SELF_TEST;
                            else if(last_data_received == 8'b1111_0010)
    // Device ID request
                                state <= SEND_DEVICE_ID;
                            else
                                state <= IDLE;

                            keyboard_buffer_enqueue_data <= 1;
                            keyboard_buffer_data_to_enqueue <=
24'b1111_1010_0000_0000_0000_0000;

                            keyboard_buffer_clear <= 0;
                            serial_buffer_clear <= 0;
                        end

                    // Send device ID
                    SEND_DEVICE_ID:
```

```
                    begin
                        state <= IDLE;

                        keyboard_buffer_enqueue_data <= 1;
                        keyboard_buffer_data_to_enqueue <=
24'b1010_1011_1000_0011_0000_0000;

                        keyboard_buffer_clear <= 0;
                        serial_buffer_clear <= 0;
                    end

                // Send echo
                SEND_ECHO:
                    begin
                        state <= IDLE;

                        keyboard_buffer_enqueue_data <= 1;
                        keyboard_buffer_data_to_enqueue <=
24'b1110_1110_0000_0000_0000_0000;

                        keyboard_buffer_clear <= 0;
                        serial_buffer_clear <= 0;
                    end
            endcase
        end
endmodule
```

ps2_deserializer.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
////////////
//
// Ceryen Tan
//
// Deserializes packets from the PS/2 data line
//
//////////////////////////////////////////////////////////////////////////////////
////////////
module ps2_deserializer(reset, clk_27_mhz,
                            ps2_clock_in, ps2_data_in, ps2_inhibit,
                            ps2_data_out, clock_gen_clock_enable,
                            received_data, data_received,
receive_error);
    input reset;
    input clk_27_mhz;

    input ps2_clock_in;
    input ps2_data_in;
    input ps2_inhibit;

    output ps2_data_out;
    output clock_gen_clock_enable;

    output received_data;
```

```
output [7:0] data_received;
output receive_error;

parameter NUM_CYCLES_CLOCK_HIGH_BEFORE_SEND = 500;

parameter IDLE = 0;
parameter WAIT_FOR_REQUEST_TO_SEND = 1;
parameter RECEIVE_CLOCK_HIGH = 2;
parameter RECEIVE_CLOCK_LOW = 3;
parameter START_SEND_ACK = 4;
parameter END_SEND_ACK = 5;

reg [2:0] state;
reg [9:0] clock_high_count;

reg ps2_data_out;
reg clock_gen_clock_enable;

reg [3:0] num_bits_received;
reg [8:0] received_data_shift_regs;

reg received_data;
reg [7:0] data_received;
reg receive_error;

initial
    begin
        state = IDLE;
        clock_high_count = 0;

        ps2_data_out = 1;
        clock_gen_clock_enable = 0;

        num_bits_received = 0;
        received_data_shift_regs = 0;

        received_data = 0;
        data_received = 0;
        receive_error = 0;
    end

always @(posedge clk_27_mhz)
    begin
        clock_high_count <= 0;

        ps2_data_out <= 1;
        clock_gen_clock_enable <= 0;

        num_bits_received <= 0;
        received_data_shift_regs <= 0;

        received_data <= 0;
        data_received <= 0;
        receive_error <= 0;

        if(reset)
            state <= IDLE;
```

```
            else
                case(state)
                    IDLE:
                        if(ps2_inhibit)
                            state <= WAIT_FOR_REQUEST_TO_SEND;
                        else
                            state <= IDLE;

                    WAIT_FOR_REQUEST_TO_SEND:
                        if(ps2_clock_in)
                            begin
                                if(ps2_data_in)
                                    state <= IDLE;
                                else
                                    begin
                                        state <= RECEIVE_CLOCK_HIGH;
                                        clock_gen_clock_enable <= 1;

                                        num_bits_received <= 0;
                                        received_data_shift_regs <= 0;
                                    end
                            end
                        else
                            state <= WAIT_FOR_REQUEST_TO_SEND;

                    RECEIVE_CLOCK_HIGH:
                        if(!ps2_clock_in)
                            begin
                                state <= RECEIVE_CLOCK_LOW;

                                clock_gen_clock_enable <= 1;

                                num_bits_received <= num_bits_received;
                                received_data_shift_regs[8:0] <=
received_data_shift_regs[8:0];
                            end
                        else
                            begin
                                state <= RECEIVE_CLOCK_HIGH;

                                clock_gen_clock_enable <= 1;

                                num_bits_received <= num_bits_received;
                                received_data_shift_regs[8:0] <=
received_data_shift_regs[8:0];
                            end

                    RECEIVE_CLOCK_LOW:
                        if(ps2_inhibit)
                            begin
                                state <= WAIT_FOR_REQUEST_TO_SEND;

                                clock_gen_clock_enable <= 0;

                                num_bits_received <= 0;
                                received_data_shift_regs[8:0] <= 0;
                            end
```

```
                    else if(ps2_clock_in)
                        begin
                            if(num_bits_received < 9)
                                begin
                                    state <= RECEIVE_CLOCK_HIGH;

                                    clock_gen_clock_enable <= 1;

                                    num_bits_received <=
num_bits_received + 1;
                                    received_data_shift_regs[0] <=
ps2_data_in;
                                    received_data_shift_regs[1] <=
received_data_shift_regs[0];
                                    received_data_shift_regs[2] <=
received_data_shift_regs[1];
                                    received_data_shift_regs[3] <=
received_data_shift_regs[2];
                                    received_data_shift_regs[4] <=
received_data_shift_regs[3];
                                    received_data_shift_regs[5] <=
received_data_shift_regs[4];
                                    received_data_shift_regs[6] <=
received_data_shift_regs[5];
                                    received_data_shift_regs[7] <=
received_data_shift_regs[6];
                                    received_data_shift_regs[8] <=
received_data_shift_regs[7];
                                end
                            else
                                begin
                                    state <= START_SEND_ACK;
                                    clock_high_count <= 0;

                                    clock_gen_clock_enable <= 1;

                                    num_bits_received <=
num_bits_received;
                                    received_data_shift_regs[8:0] <=
received_data_shift_regs[8:0];
                                end
                        end
                    else
                        begin
                            state <= RECEIVE_CLOCK_LOW;

                            clock_gen_clock_enable <= 1;

                            num_bits_received <= num_bits_received;
                            received_data_shift_regs[8:0] <=
received_data_shift_regs[8:0];
                        end

                START_SEND_ACK:
                    if(ps2_inhibit)
                        begin
                            state <= WAIT_FOR_REQUEST_TO_SEND;
```

```
                            clock_gen_clock_enable <= 0;

                            num_bits_received <= 0;
                            received_data_shift_regs[8:0] <= 0;
                        end
                    else if(ps2_clock_in)
                        begin
                            if(clock_high_count <
NUM_CYCLES_CLOCK_HIGH_BEFORE_SEND - 1)
                                begin
                                    state <= START_SEND_ACK;
                                    clock_high_count <=
clock_high_count + 1;

                                    clock_gen_clock_enable <= 1;

                                    num_bits_received <=
num_bits_received;
                                    received_data_shift_regs[8:0] <=
received_data_shift_regs[8:0];
                                end
                            else
                                begin
                                    state <= START_SEND_ACK;
                                    clock_high_count <=
clock_high_count;

                                    ps2_data_out <= 0;
                                    clock_gen_clock_enable <= 1;

                                    num_bits_received <=
num_bits_received;
                                    received_data_shift_regs[8:0] <=
received_data_shift_regs[8:0];
                                end
                            end
                        else
                            begin
                                state <= END_SEND_ACK;
                                clock_high_count <= 0;

                                ps2_data_out <= 0;
                                clock_gen_clock_enable <= 1;

                                num_bits_received <= num_bits_received;
                                received_data_shift_regs[8:0] <=
received_data_shift_regs[8:0];
                            end

                    END_SEND_ACK:
                        if(ps2_inhibit)
                            begin
                                state <= WAIT_FOR_REQUEST_TO_SEND;

                                clock_gen_clock_enable <= 0;
```

```
                            num_bits_received <= 0;
                            received_data_shift_regs[8:0] <= 0;
                        end
                    else if(!ps2_clock_in)
                        begin
                            state <= END_SEND_ACK;
                            clock_high_count <= 0;

                            ps2_data_out <= 0;
                            clock_gen_clock_enable <= 1;

                            num_bits_received <= num_bits_received;
                            received_data_shift_regs[8:0] <=
received_data_shift_regs[8:0];
                        end
                    else if(clock_high_count <
NUM_CYCLES_CLOCK_HIGH_BEFORE_SEND - 1)
                            begin
                            state <= END_SEND_ACK;
                            clock_high_count <= clock_high_count + 1;

                            ps2_data_out <= 0;
                            clock_gen_clock_enable <= 1;

                            num_bits_received <= num_bits_received;
                            received_data_shift_regs[8:0] <=
received_data_shift_regs[8:0];
                            end
                    else
                        begin
                            state <= IDLE;

                            received_data <= 1;
                            data_received[0] <=
received_data_shift_regs[8];
                            data_received[1] <=
received_data_shift_regs[7];
                            data_received[2] <=
received_data_shift_regs[6];
                            data_received[3] <=
received_data_shift_regs[5];
                            data_received[4] <=
received_data_shift_regs[4];
                            data_received[5] <=
received_data_shift_regs[3];
                            data_received[6] <=
received_data_shift_regs[2];
                            data_received[7] <=
received_data_shift_regs[1];
                            receive_error <=
                                (((received_data_shift_regs[8] ^
received_data_shift_regs[7]) ^
                                  (received_data_shift_regs[6] ^
received_data_shift_regs[5])) ^
                                   ((received_data_shift_regs[4] ^
received_data_shift_regs[3]) ^
```

```
                                    (received_data_shift_regs[2] ^
received_data_shift_regs[1]))) ^
                                received_data_shift_regs[0];
                        end
                endcase
        end
endmodule
```

ps2_inhibit_detect.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////
////////////
//
// Ceryen Tan
//
// When the PS/2 clock is pulled low for 100 us, this means that the
host is
// inhibiting communication between the device and the host.  No
information
// can be sent over the PS/2 interface when communication is inhibited.
//
// The host may inhibit communication for a number of reasons:
// 1) The host may inhibit communication to signal host-to-device
//    communication.  To initiate host-to-device communication, the
host first
//    inhibits communication by pulling clock low for 100 us.  The host
then
//    pulls data low and releases clock.  This signals to the device
that the
//    host wants to transmit information.  This sequence of events must
start
//    off with the host inhibiting communication.
// 2) The host may inhibit communication when it is processing
information snet
//    from the keyboard.  For example, the host might inhibit
communication
//    after every byte sent from the keyboard.
// 3) Communication can be inhibited at any arbitrary time for any
arbitrary
//    reason.  This actually occurs quite often.
//
// The purpose of this module is to simply detect when communication is
// inhibited.  When clock is pulled low for more than 100 uS, this
module
// outputs an inhibit signal to tell other PS/2 modules that
communcation
// is inhibited.
//
//////////////////////////////////////////////////////////////////////////////
////////////
module ps2_inhibit_detect(reset, clk_27_mhz, ps2_clock_in,
ps2_inhibit);
    input reset;
```

```
    input clk_27_mhz;
    input ps2_clock_in;
    output ps2_inhibit;

    // The number of 27 MHz clock cycles that correspond to 100 us in
time.
    // This parameter determines how long the PS/2 clock must be pulled
low for
    // the inhibit signal to be asserted
    parameter NUM_CLOCK_LOW_FOR_INHIBIT = 2700;

    // Counts the number of 27 MHz clock cycles that the PS/2 clock is
low
    reg [11:0] clock_low_count;

    // Output for whether or not communication is inhibited
    reg ps2_inhibit;

    always @(posedge clk_27_mhz)
        begin
            // If reset or PS/2 clock is high, reset the count for how
many cycles
            // the PS/2 clock has been low and deassert the inhibit
signal
            if(reset || ps2_clock_in)
                begin
                    clock_low_count <= 0;
                    ps2_inhibit <= 0;
                end

            // If the above if statement is not true, then PS/2 clock is
presently
            // low.  If communication is already inhibited, then continue
            // asserting the inhibit signal.  Reset the low clock count
since it's
            // pointless to keep counting
            else if(ps2_inhibit)
                begin
                    clock_low_count <= 0;
                    ps2_inhibit <= 1;
                end

            // If not yet inhibited, check to see if PS/2 clock has been
low long
            // enough to assert the inhibit signal.  If so, then assert
the
            // inhibit signal and stop counting since it is pointless to
continue
            else if(clock_low_count == NUM_CLOCK_LOW_FOR_INHIBIT - 1)
                begin
                    clock_low_count <= 0;
                    ps2_inhibit <= 1;
                end

            // Not yet inhibited and PS/2 clock is low.  Count the number
of 27
            // MHz clock cycle that the PS/2 clock is low
```

```
            else
                begin
                    clock_low_count <= clock_low_count + 1;
                    ps2_inhibit <= 0;
                end
        end
endmodule
```

ps_keyboard.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
//////////
//
// Ceryen Tan
//
// This module packages all of the other keyboard modules into a single
module
//
//////////////////////////////////////////////////////////////////////////////////
//////////
module ps2_keyboard(manual_reset, clk_27_mhz,
                        ps2_power,
                        ps2_data_in, ps2_data_out,
                        ps2_clock_in, ps2_clock_out,
                        action_enable_0,
                        action_enable_1,
                        action_enable_2,
                        action_enable_3,
                        action_enable_4,
                        action_enable_5,
                        action_enable_6,
                        action_enable_7,
                        action_make_code_0,
                        action_make_code_1,
                        action_make_code_2,
                        action_make_code_3,
                        action_make_code_4,
                        action_make_code_5,
                        action_make_code_6,
                        action_make_code_7);

    input manual_reset;
    input clk_27_mhz;

    input ps2_power;

    input ps2_data_in;
    output ps2_data_out;

    input ps2_clock_in;
    output ps2_clock_out;

    input action_enable_0;
    input action_enable_1;
```

```
    input action_enable_2;
    input action_enable_3;
    input action_enable_4;
    input action_enable_5;
    input action_enable_6;
    input action_enable_7;

    input [15:0] action_make_code_0;
    input [15:0] action_make_code_1;
    input [15:0] action_make_code_2;
    input [15:0] action_make_code_3;
    input [15:0] action_make_code_4;
    input [15:0] action_make_code_5;
    input [15:0] action_make_code_6;
    input [15:0] action_make_code_7;

    // Reset detection
    wire reset_due_to_power;
    ps2_power_detect ps2_power_detect(clk_27_mhz, ps2_power,
reset_due_to_power);

    wire reset = manual_reset || reset_due_to_power;

    // Inhibit detector
    wire ps2_inhibit;

    ps2_inhibit_detect ps2_inhibit_detect(reset, clk_27_mhz,
ps2_clock_in, ps2_inhibit);

    // Clock generator
    wire clock_gen_clock_enable;

    wire clock_gen_clock_enable_serializer;
    wire clock_gen_clock_enable_deserializer;

    assign clock_gen_clock_enable = clock_gen_clock_enable_serializer ||

clock_gen_clock_enable_deserializer;

    ps2_clock_gen ps2_clock_gen(
        .reset(reset),
        .clk_27_mhz(clk_27_mhz),
        .ps2_clock_enable(clock_gen_clock_enable),
        .ps2_clock_out(ps2_clock_out)
    );

    // Data out wire
    wire ps2_data_out_serializer;
    wire ps2_data_out_deserializer;

    assign ps2_data_out = ps2_data_out_serializer &&
ps2_data_out_deserializer;

    // Serializer
    wire serial_send_data;
    wire [7:0] serial_data_to_send;
```

```
wire serial_send_success;
wire serial_send_failure;
wire serial_line_idle;

ps2_serializer ps2_serializer(
    .reset(reset),
    .clk_27_mhz(clk_27_mhz),
    .ps2_clock_in(ps2_clock_in),
    .ps2_inhibit(ps2_inhibit),
    .ps2_data_out(ps2_data_out_serializer),
    .clock_gen_clock_enable(clock_gen_clock_enable_serializer),
    .serial_send_data(serial_send_data),
    .serial_data_to_send(serial_data_to_send),
    .serial_line_idle(serial_line_idle),
    .serial_send_success(serial_send_success),
    .serial_send_failure(serial_send_failure)
);

// Serializer buffer
wire serial_buffer_clear;
wire serial_buffer_send_data;
wire [23:0] serial_buffer_data_to_send;

wire serial_buffer_idle;

ps2_serializer_buffer ps2_serializer_buffer(
    .reset(reset),
    .clk_27_mhz(clk_27_mhz),
    .serial_buffer_clear(serial_buffer_clear),
    .serial_buffer_send_data(serial_buffer_send_data),
    .serial_buffer_data_to_send(serial_buffer_data_to_send),
    .serial_buffer_idle(serial_buffer_idle),
    .serial_send_data(serial_send_data),
    .serial_data_to_send(serial_data_to_send),
    .serial_line_idle(serial_line_idle),
    .serial_send_success(serial_send_success),
    .serial_send_failure(serial_send_failure)
);

// Keyboard buffer
wire keyboard_buffer_clear;
wire keyboard_buffer_enqueue_data;
wire [23:0] keyboard_buffer_data_to_enqueue;

wire keyboard_buffer_enqueue_data_control_fsm;
wire [23:0] keyboard_buffer_data_to_enqueue_control_fsm;

wire keyboard_buffer_enqueue_data_ext;
wire [23:0] keyboard_buffer_data_to_enqueue_ext;

assign keyboard_buffer_enqueue_data =
    keyboard_buffer_enqueue_data_control_fsm ||
    keyboard_buffer_enqueue_data_ext;
assign keyboard_buffer_data_to_enqueue =
    keyboard_buffer_enqueue_data_control_fsm ?
        keyboard_buffer_data_to_enqueue_control_fsm :
        keyboard_buffer_data_to_enqueue_ext;
```

```
    ps2_keyboard_buffer ps2_keyboard_buffer(
        .reset(reset),
        .clk_27_mhz(clk_27_mhz),
        .keyboard_buffer_clear(keyboard_buffer_clear),
        .keyboard_buffer_enqueue_data(keyboard_buffer_enqueue_data),

    .keyboard_buffer_data_to_enqueue(keyboard_buffer_data_to_enqueue),
        .serial_buffer_send_data(serial_buffer_send_data),
        .serial_buffer_data_to_send(serial_buffer_data_to_send),
        .serial_buffer_idle(serial_buffer_idle)
    );

    // Action-to-key typematic converter
    ps2_typematic_converter ps2_typematic_converter(
        .reset(reset),
        .clk_27_mhz(clk_27_mhz),
        .action_enable_0(action_enable_0),
        .action_enable_1(action_enable_1),
        .action_enable_2(action_enable_2),
        .action_enable_3(action_enable_3),
        .action_enable_4(action_enable_4),
        .action_enable_5(action_enable_5),
        .action_enable_6(action_enable_6),
        .action_enable_7(action_enable_7),
        .action_make_code_0(action_make_code_0),
        .action_make_code_1(action_make_code_1),
        .action_make_code_2(action_make_code_2),
        .action_make_code_3(action_make_code_3),
        .action_make_code_4(action_make_code_4),
        .action_make_code_5(action_make_code_5),
        .action_make_code_6(action_make_code_6),
        .action_make_code_7(action_make_code_7),
        .keyboard_buffer_enqueue_data(keyboard_buffer_enqueue_data_ext),

    .keyboard_buffer_data_to_enqueue(keyboard_buffer_data_to_enqueue_ext
)
    );

    // Deserializer
    wire deserial_received_data;
    wire [7:0] deserial_data_received;
    wire deserial_receive_error;

    ps2_deserializer ps2_deserializer(
        .reset(reset),
        .clk_27_mhz(clk_27_mhz),
        .ps2_clock_in(ps2_clock_in),
        .ps2_data_in(ps2_data_in),
        .ps2_inhibit(ps2_inhibit),
        .ps2_data_out(ps2_data_out_deserializer),
        .clock_gen_clock_enable(clock_gen_clock_enable_deserializer),
        .received_data(deserial_received_data),
        .data_received(deserial_data_received),
        .receive_error(deserial_receive_error)
    );
```

```
    // PS/2 Control FSM
    ps2_control_fsm ps2_control_fsm(
        .reset(reset),
        .clk_27_mhz(clk_27_mhz),

    .keyboard_buffer_enqueue_data(keyboard_buffer_enqueue_data_control_f
sm),

    .keyboard_buffer_data_to_enqueue(keyboard_buffer_data_to_enqueue_con
trol_fsm),
        .keyboard_buffer_clear(keyboard_buffer_clear),
        .serial_buffer_clear(serial_buffer_clear),
        .deserial_received_data(deserial_received_data),
        .deserial_data_received(deserial_data_received)
    );
endmodule
```

ps2_keyboard_buffer.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////
////////////
//
// Ceryen Tan
//
// This module temporarily buffers up to eight scan codes until they
can be
// sent over the PS/2 interface.  Scan codes can sometimes be generated
faster
// than they can be sent, which is why this module is necessarily.
//
// The keyboard buffer outputs a message to the serializer buffer when
the
// serializer buffer is not actively sending a message
//
//////////////////////////////////////////////////////////////////////////
////////////
module ps2_keyboard_buffer(reset, clk_27_mhz,
                                keyboard_buffer_clear,
                                keyboard_buffer_enqueue_data,
keyboard_buffer_data_to_enqueue,
                                serial_buffer_send_data,
serial_buffer_data_to_send,
                                serial_buffer_idle);
    input reset;
    input clk_27_mhz;

    // Clears keyboard buffer
    input keyboard_buffer_clear;

    // Enqueues a new scancode into the buffer
    input keyboard_buffer_enqueue_data;
    input [23:0] keyboard_buffer_data_to_enqueue;
```

```
    // Sends a scancode to the serializer buffer when the serializer
buffer is idle
    output serial_buffer_send_data;
    output [23:0] serial_buffer_data_to_send;

    input serial_buffer_idle;

    // Actual registers that stores scan codes
    reg [191:0] keyboard_buffer;

    // The number of scan codes that is presently stored in the keyboard
buffer
    reg [3:0] num_stored_packets;

    // Sends a scancode to the serializer buffer when the serializer
buffer is idle
    reg serial_buffer_send_data;
    reg [23:0] serial_buffer_data_to_send;

    // This is necessary because there is a one cycle delay between the
sending
    // the serializer buffer new data and the buffer reporting that it
is busy
    reg skip_next_buffer_idle;

    initial
        begin
            keyboard_buffer <= 192'b0;
            num_stored_packets <= 0;

            serial_buffer_send_data <= 0;
            serial_buffer_data_to_send <= 24'b0;

            skip_next_buffer_idle <= 0;
        end

    always @(posedge clk_27_mhz)
        begin
            serial_buffer_send_data <= 0;
            serial_buffer_data_to_send <= 24'b0;

            skip_next_buffer_idle <= 0;

            if(reset || keyboard_buffer_clear)
                begin
                    keyboard_buffer <= 192'b0;
                    num_stored_packets <= 4'b0;
                end
            else if(keyboard_buffer_enqueue_data)
                // Keyboard buffer told to enqueue new scan code
                begin
                    keyboard_buffer[191:0] <= keyboard_buffer[191:0];

                    // Copy new scan code into buffer wherever there is
room
                    case(num_stored_packets)
```

```
                    0: keyboard_buffer[191:168] <=
keyboard_buffer_data_to_enqueue;
                    1: keyboard_buffer[167:144] <=
keyboard_buffer_data_to_enqueue;
                    2: keyboard_buffer[143:120] <=
keyboard_buffer_data_to_enqueue;
                    3: keyboard_buffer[119:96] <=
keyboard_buffer_data_to_enqueue;
                    4: keyboard_buffer[95:72] <=
keyboard_buffer_data_to_enqueue;
                    5: keyboard_buffer[71:48] <=
keyboard_buffer_data_to_enqueue;
                    6: keyboard_buffer[47:24] <=
keyboard_buffer_data_to_enqueue;
                    7: keyboard_buffer[23:0] <=
keyboard_buffer_data_to_enqueue;
                endcase

                // Increment the number of stored scan codes
                if(num_stored_packets < 8)
                    num_stored_packets <= num_stored_packets + 1;
                else
                    num_stored_packets <= 8;
            end
        else if(serial_buffer_idle && !skip_next_buffer_idle)
            // If the serializer buffer is ready to send a new scan
code, give
            // it a new scan code to send
            begin
                // If there is a scancode to send
                if(num_stored_packets != 0)
                    begin
                        // Send scan code
                        serial_buffer_send_data <= 1;
                        serial_buffer_data_to_send <=
keyboard_buffer[191:168];

                        // Remove scan code from buffer
                        keyboard_buffer[191:168] <=
keyboard_buffer[167:144];
                        keyboard_buffer[167:144] <=
keyboard_buffer[143:120];
                        keyboard_buffer[143:120] <=
keyboard_buffer[119:96];
                        keyboard_buffer[119:96] <=
keyboard_buffer[95:72];
                        keyboard_buffer[95:72] <=
keyboard_buffer[71:48];
                        keyboard_buffer[71:48] <=
keyboard_buffer[47:24];
                        keyboard_buffer[47:24] <=
keyboard_buffer[23:0];
                        keyboard_buffer[23:0] <= 24'b0;

                        // Decrement the number of scan codes stored
in the buffer
                        num_stored_packets <= num_stored_packets - 1;
```

```
                        // This is necessary because there is a one
cycle
                        // delay between the sending the serializer
buffer
                        // new data and the buffer reporting that it
is busy
                        skip_next_buffer_idle <= 1;
                    end
                else
                    begin
                        keyboard_buffer <= 192'b0;
                        num_stored_packets <= 4'b0;
                    end
            end
        else
            begin
                keyboard_buffer <= keyboard_buffer;
                num_stored_packets <= num_stored_packets;

                if(skip_next_buffer_idle)
                    skip_next_buffer_idle <= 0;
            end
    end
endmodule
```

ps2_key_event_detector.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////
//////////
//
// Ceryen Tan
//
// Generates pressed and released events from a key enable signal
//
//////////////////////////////////////////////////////////////////////
//////////
module ps2_key_event_detector(reset, clk_27_mhz, key_input,
key_pressed, key_released);
    input reset;
    input clk_27_mhz;
    input key_input;
    output key_pressed;
    output key_released;

    reg last_key_input;

    reg key_pressed;
    reg key_released;

    always @(posedge clk_27_mhz)
        begin
            last_key_input <= key_input;
```

```
            if(reset)
                begin
                    last_key_input <= 0;

                    key_pressed <= 0;
                    key_released <= 0;
                end
            else if(last_key_input == 0 && key_input == 1)
                begin
                    key_pressed <= 1;
                    key_released <= 0;
                end
            else if(last_key_input == 1 && key_input == 0)
                begin
                    key_pressed <= 0;
                    key_released <= 1;
                end
            else
                begin
                    key_pressed <= 0;
                    key_released <= 0;
                end
        end
endmodule
```

ps2_power_detect.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
////////////
//
// Ceryen Tan
//
// This module detects when power is supplied to the PS/2 interface and
// generates a reset command.  This is so that when the computer is
initially
// turned on, the PS/2 interface starts out without any state.  This
also
// allows any initialization procedures to run on computer startup.
//
//////////////////////////////////////////////////////////////////////////////////
////////////
module ps2_power_detect(clk_27_mhz, ps2_power, reset_due_to_power);
    input clk_27_mhz;
    input ps2_power;
    output reset_due_to_power;

    // Store the previous power state
    reg prev_power_state;

    // Output a reset signal when the power turns on
    reg reset_due_to_power;

    initial
        begin
```

```
            prev_power_state <= 0;
            reset_due_to_power <= 0;
        end

    always @(posedge clk_27_mhz)
        // If computer power switches from off to on, then generate a
reset signal
        if(prev_power_state == 0 && ps2_power == 1)
            begin
                prev_power_state <= 1'b1;
                reset_due_to_power <= 1;
            end

        // Otherwise, store the current power state and don't generate a
reset
        else
            begin
                prev_power_state <= ps2_power;
                reset_due_to_power <= 0;
            end
endmodule
```

ps2_scancode_buffer.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////
////////////
//
// Ceryen Tan
//
// This module wraps around the serializer in order to allow the
transmission
// of messages that take up multiple bytes.  This module feeds in
message bytes
// to the serializer one byte at a time and monitors whether or not
each byte
// is sent successfully.  If any one message byte is sent
unsuccessfully, then
// this module restarts the message by resending from the first byte.
//
// This module allows the sending of bytes that are up to three bytes
long.
// Messages are passed in using the serial_buffer_send_data and
// serial_buffer_data_to_send
//
// This module outputs to serial_buffer_idle to signal whether it is
busy
// sending a multiple byte message
//
//////////////////////////////////////////////////////////////////////
////////////
module ps2_serializer_buffer(reset, clk_27_mhz,
                                    serial_buffer_clear,
                                    serial_buffer_send_data,
serial_buffer_data_to_send,
```

```
                                    serial_buffer_idle,
                                    serial_send_data,
serial_data_to_send,
                                    serial_line_idle,
serial_send_success, serial_send_failure);
    input reset;
    input clk_27_mhz;

    // Clears whatever message the buffer is trying to send
    input serial_buffer_clear;

    // Inputs a new message to send
    input serial_buffer_send_data;
    input [23:0] serial_buffer_data_to_send;

    // Whether or not the serializer buffer is busy sending a message
    output serial_buffer_idle;

    // Interaction with the serializer module
    output serial_send_data;
    output [7:0] serial_data_to_send;

    input serial_line_idle;
    input serial_send_success;
    input serial_send_failure;

    // FSM states
    parameter IDLE = 0;                     // Not sending a message

    parameter WAIT_SEND_BYTE_1 = 1;     // Sending of byte 1
    parameter SEND_BYTE_1 = 2;
    parameter WAIT_FINISH_BYTE_1 = 3;

    parameter WAIT_SEND_BYTE_2 = 4;     // Sending of byte 2
    parameter SEND_BYTE_2 = 5;
    parameter WAIT_FINISH_BYTE_2 = 6;

    parameter WAIT_SEND_BYTE_3 = 7;     // Sending of byte 3
    parameter SEND_BYTE_3 = 8;
    parameter WAIT_FINISH_BYTE_3 = 9;

    reg [3:0] state;

    // Stores the multiple byte message to send over the PS/2 interface
    reg [23:0] data_to_send;

    // Interaction with the serializer module to send individual bytes
at a time
    reg serial_send_data;
    reg [7:0] serial_data_to_send;

    // Whether or not the serializer buffer is presently busy sending a
message
    reg serial_buffer_idle;

    initial
        begin
```

```
            state <= IDLE;

            data_to_send <= 24'b0;

            serial_send_data <= 0;
            serial_data_to_send <= 8'b0;

            serial_buffer_idle <= 1;
        end

    always @(posedge clk_27_mhz)
        begin
            data_to_send <= 24'b0;

            serial_send_data <= 0;
            serial_data_to_send <= 8'b0;

            serial_buffer_idle <= 1;

            if(reset || serial_buffer_clear)
                state <= IDLE;
            else
                case(state)
                    IDLE:
                        // If receive new message to send, copy it down
and start
                        // sending it
                        if(serial_buffer_send_data)
                            begin
                                state <= WAIT_SEND_BYTE_1;
                                data_to_send <= serial_buffer_data_to_send;

                                serial_buffer_idle <= 0;
                            end
                        else
                            begin
                                state <= IDLE;
                                data_to_send <= 24'b0;

                                serial_buffer_idle <= 1;
                            end

                    WAIT_SEND_BYTE_1:
                        // If the serializer is idle, send it the first
byte
                        if(serial_line_idle)
                            begin
                                state <= SEND_BYTE_1;
                                data_to_send <= data_to_send;

                                serial_send_data <= 1;
                                serial_data_to_send <= data_to_send[23:16];

                                serial_buffer_idle <= 0;
                            end

                        // Else, wait until the serializer is idle
```

```
                    else
                        begin
                            state <= WAIT_SEND_BYTE_1;
                            data_to_send <= data_to_send;

                            serial_buffer_idle <= 0;
                        end

                SEND_BYTE_1:
                    // If serializer has not yet starting sending
first byte,
                    // keep sending first byte to serializer
                    if(serial_line_idle)
                        begin
                            state <= SEND_BYTE_1;
                            data_to_send <= data_to_send;

                            serial_send_data <= 1;
                            serial_data_to_send <= data_to_send[23:16];

                            serial_buffer_idle <= 0;
                        end
                    else
                        begin
                            state <= WAIT_FINISH_BYTE_1;
                            data_to_send <= data_to_send;

                            serial_buffer_idle <= 0;
                        end

                WAIT_FINISH_BYTE_1:
                    // If first byte sent successfully, move to second
                    if(serial_send_success)
                        begin
                            // Check if there is a second byte to send
                            if(data_to_send[15:8] != 8'b0)
                                begin
                                    state <= WAIT_SEND_BYTE_2;
                                    data_to_send <= data_to_send;

                                    serial_buffer_idle <= 0;
                                end
                            else
                                begin
                                    state <= IDLE;
                                    data_to_send <= 24'b0;

                                    serial_buffer_idle <= 1;
                                end
                        end

                    // If first byte failed, restart completely
                    else if(serial_send_failure)
                        begin
                            state <= WAIT_SEND_BYTE_1;
                            data_to_send <= data_to_send;
```

```
                        serial_buffer_idle <= 0;
                    end
                else
                    begin
                        state <= WAIT_FINISH_BYTE_1;
                        data_to_send <= data_to_send;

                        serial_buffer_idle <= 0;
                    end

            WAIT_SEND_BYTE_2:
                // If serializer is now idle, send it the 2nd byte
                if(serial_line_idle)
                    begin
                        state <= SEND_BYTE_2;
                        data_to_send <= data_to_send;

                        serial_send_data <= 1;
                        serial_data_to_send <= data_to_send[15:8];

                        serial_buffer_idle <= 0;
                    end

                // Else wait for it to be idle
                else
                    begin
                        state <= WAIT_SEND_BYTE_2;
                        data_to_send <= data_to_send;

                        serial_buffer_idle <= 0;
                    end

            SEND_BYTE_2:
                // If serializer has not yet starting sending
second byte,
                // keep sending it the second byte
                if(serial_line_idle)
                    begin
                        state <= SEND_BYTE_2;
                        data_to_send <= data_to_send;

                        serial_send_data <= 1;
                        serial_data_to_send <= data_to_send[15:8];

                        serial_buffer_idle <= 0;
                    end

                // Else stop sending it the second byte
                else
                    begin
                        state <= WAIT_FINISH_BYTE_2;
                        data_to_send <= data_to_send;

                        serial_buffer_idle <= 0;
                    end

            WAIT_FINISH_BYTE_2:
```

```
                    // If second byte sent successfully, move to third
byte
                    if(serial_send_success)
                        begin
                            // Check if there is a third byte to send
                            if(data_to_send[7:0] != 8'b0)
                                begin
                                    state <= WAIT_SEND_BYTE_3;
                                    data_to_send <= data_to_send;

                                    serial_buffer_idle <= 0;
                                end
                            else
                                begin
                                    state <= IDLE;
                                    data_to_send <= 24'b0;

                                    serial_buffer_idle <= 1;
                                end
                        end

                    // If second byte failed, restart completely
                    else if(serial_send_failure)
                        begin
                            state <= WAIT_SEND_BYTE_1;
                            data_to_send <= data_to_send;

                            serial_buffer_idle <= 0;
                        end
                    else
                        begin
                            state <= WAIT_FINISH_BYTE_2;
                            data_to_send <= data_to_send;

                            serial_buffer_idle <= 0;
                        end

                WAIT_SEND_BYTE_3:
                    // If serializer is idle, send it third byte
                    if(serial_line_idle)
                        begin
                            state <= SEND_BYTE_3;
                            data_to_send <= data_to_send;

                            serial_send_data <= 1;
                            serial_data_to_send <= data_to_send[7:0];

                            serial_buffer_idle <= 0;
                        end

                    // Else, wait until serializer is idle
                    else
                        begin
                            state <= WAIT_SEND_BYTE_3;
                            data_to_send <= data_to_send;

                            serial_buffer_idle <= 0;
```

```
                        end

                    SEND_BYTE_3:
                        // If not yet started sending, keep sending
serializer third byte
                        if(serial_line_idle)
                            begin
                                state <= SEND_BYTE_3;
                                data_to_send <= data_to_send;

                                serial_send_data <= 1;
                                serial_data_to_send <= data_to_send[7:0];

                                serial_buffer_idle <= 0;
                            end

                        // Else, wait until for third byte to finish
sending
                        else
                            begin
                                state <= WAIT_FINISH_BYTE_3;
                                data_to_send <= data_to_send;

                                serial_buffer_idle <= 0;
                            end

                    WAIT_FINISH_BYTE_3:
                        // If third byte successful, quit
                        if(serial_send_success)
                            begin
                                state <= IDLE;
                                data_to_send <= 24'b0;

                                serial_buffer_idle <= 1;
                            end

                        // Else, restart completely
                        else if(serial_send_failure)
                            begin
                                state <= WAIT_SEND_BYTE_1;
                                data_to_send <= data_to_send;

                                serial_buffer_idle <= 0;
                            end
                        else
                            begin
                                state <= WAIT_FINISH_BYTE_3;
                                data_to_send <= data_to_send;

                                serial_buffer_idle <= 0;
                            end
                endcase
        end
endmodule
```

ps2_serializer.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
//
// Ceryen Tan
//
// This module is responsible for serializing data over the PS/2 data
line.
// This module is capable of transmitting a byte of data at a time.
//
// Data can be passed into this module to be serialized onto the the
PS/2 data
// line using the serial_send_data and serial_data_to_send inputs.
When
// serial_send_data is asserted high, serial_data_to_send is copied
over to be
// transmitted over the PS/2 data line.  These two inputs only need to
be
// asserted for a single 27 MHz clock cycle.
//
// In order to serialize data over the PS/2 data line, a PS/2 clock
signal must
// be generated.  This module asserts clock_gen_clock_enable high when
it is
// serializing data so that the PS/2 clock generator will generate a
PS/2 clock
// signal.  This module assumes that the PS/2 clock generator will
first start
// with a high pules.
//
// When the serializer is not sending data over the data line, the PS/2
data
// line is left high.  This is similar to the PS/2 clock line, which is
also
// left high by default.
//
// Note that in order for data to be transmitted over the PS/2 data
line, the
// PS/2 clock line must be high for at least 50 us before transmission.
This
// module keeps track of how long the PS/2 clock line is high.  If the
PS/2
// clock has been high for at least 50 us, this module asserts
serial_line_idle
// high to show that the PS/2 data line is idle and ready to transmit
data.
// If the PS/2 clock has not been high for 50 us, serial_line_idle is
set low
// to show that the PS/2 data line is not idle and not ready to
transmit data.
//
// If another module attempts to transmit data when the data line is
not idle,
// then this module ignores the inputted data.  In other words, if
```

```
// serial_send_data is asserted high while serial_line_idle is low,
this module
// ignores any input data.  This input data is not transmitted over the
PS/2
// data line.
//
// Note that the host may choose to inhibit communication at any time,
even
// while the device is serializing data onto the PS/2 data line.  If
// communication is inhibited while a byte is being transmitted, then
// transmission of the byte fails.  This module asserts
serial_send_failure
// high for one clock cycle when this happens, and gives up on sending
the byte
// of data.
//
// Conversely, if a byte of data is successfully sent (i.e.
communication is
// not inhibited while the data is transmitted), then
serial_send_success is
// asserted high for one clock cycle.
//
// A byte of data is packaged into a single packet of 11 bits.  This
packet
// contains:
// 1) Start bit (always 0)
// 2-9) Data bits
// 10) Odd parity bit
// 11) Stop bit (always 1)
//
// Serializing data onto the PS/2 data line is required to meet a
specific
// timing specification.  Most other 6.111 groups chose to ignore this
timing
// specification for simplicity.  However, this timing specification is
// implemented here.
//
// Data sent from the device to the host is read on the falling edge of
the
// PS/2 clock.  This means that data is shifted out onto the PS/2 data
line
// when the PS/2 clock is high and held stable while the PS/2 clock is
low.
//
// The time between a data transition to the falling edge of the PS/2
clock
// must be between 5 us and 25 us.  To be completely safe, data
transitions
// should occur in the middle of the PS/2 clock high cycle.
//
// Because the clock generator generates a 13.5 kHz signal, the PS/2
clock is
// high for 1000 cycles of the 27 MHz clock.  This serializer shifts
out new
// data bits every time that the PS/2 clock is high for 500 cycles.
This
```

```
// creates a data transition in the middle of the PS/2 clock high
cycle.
//
//////////////////////////////////////////////////////////////////////
///////////
module ps2_serializer(reset, clk_27_mhz,
                          ps2_clock_in, ps2_inhibit,
                          ps2_data_out, clock_gen_clock_enable,
                          serial_send_data, serial_data_to_send,
                          serial_line_idle, serial_send_success,
serial_send_failure);
    input reset;
    input clk_27_mhz;

    // PS/2 clock signal
    input ps2_clock_in;

    // Whether or not communication has been inhibited
    input ps2_inhibit;

    // Output data signal
    output ps2_data_out;

    // Output that turns on PS/2 clock generator when serializing data
onto the
    // PS/2 data line
    output clock_gen_clock_enable;

    // Takes in a new byte to send
    input serial_send_data;
    input [7:0] serial_data_to_send;

    // Outputs the current status of the serializer (i.e. if the data
line is
    // idle, if a data byte has been sent successfully, if a data byte
has not
    // been sent successfully)
    output serial_line_idle;
    output serial_send_success;
    output serial_send_failure;

    // Number of 27 MHz clock cycles that the PS/2 clock signal must be
high for
    // the data line to be considered idle and ready for sending data
    parameter NUM_CYCLES_CLOCK_HIGH_FOR_IDLE = 1350;

    // Number of 27 MHz clock cycles that the PS/2 clock signal must be
high
    // before the serializer shifts out a new bit
    parameter NUM_CYCLES_CLOCK_HIGH_BEFORE_SEND = 500;

    // FSM states for the serializer
    parameter LINE_NOT_IDLE = 0;              // Not serializing data,
data line is not idle
    parameter LINE_IDLE = 1;                  // Not serializing data,
data line is idle
```

```
    parameter SENDING_PS2_CLOCK_HIGH = 2;      // Serializing data, PS/2
clock is high
    parameter SENDING_PS2_CLOCK_LOW = 3;       // Serializing data, PS/2
clock is low

    reg [1:0] state;                              // Current FSM state
    reg [31:0] clock_high_timer;               // 27 MHz clock timer

    reg clock_gen_clock_enable;                // Turns on/off PS/2
clock generator

    reg ps2_data_out;                             // Output data signal
    reg [3:0] num_bits_sent;                   // Number of bits already
sent
    reg [10:0] data_to_send_shift_regs;        // Shift registers used
to shift data onto data line

    reg serial_line_idle;                      // If data line is
presently idle
    reg serial_send_success;                   // If byte sent
successfully
    reg serial_send_failure;                   // If byte not sent
successfully

    initial
        begin
            state = LINE_NOT_IDLE;
            clock_high_timer = 0;

            clock_gen_clock_enable = 0;

            ps2_data_out = 1;
            num_bits_sent[3:0] = 0;
            data_to_send_shift_regs[10:0] = 0;

            serial_line_idle = 0;
            serial_send_success = 0;
            serial_send_failure = 0;
        end

    always @(posedge clk_27_mhz)
        begin
            // Default values
            clock_high_timer <= 0;

            clock_gen_clock_enable <= 0;           // By default, clock
gen is off

            ps2_data_out <= 1;                        // By default,
data line is high
            num_bits_sent <= 0;
            data_to_send_shift_regs[10:0] <= 0;

            serial_line_idle <= 0;                    // By default,
data line is not idle
            serial_send_success <= 0;
            serial_send_failure <= 0;
```

```
            if(reset)
                // Default state is that the line is not idle.  Clock
needs to be
                // monitored for 50 us first before declaring the line
idle
                state <= LINE_NOT_IDLE;
            else
                case(state)
                    LINE_NOT_IDLE:
                        if(ps2_clock_in)
                            // If PS/2 clock is high, count the number of
cycles that
                            // the PS/2 clock is high.  If the PS/2 clock
is high for
                            // 50 us, then declare the data line idle and
ready for
                            // transmitting
                            if(clock_high_timer ==
NUM_CYCLES_CLOCK_HIGH_FOR_IDLE - 1)
                                begin
                                    state <= LINE_IDLE;

                                    clock_high_timer <= 0;
                                    serial_line_idle <= 1;
                                end
                            else
                                begin
                                    state <= LINE_NOT_IDLE;

                                    clock_high_timer <= clock_high_timer +
1;
                                    serial_line_idle <= 0;
                                end
                        else
                            // If PS/2 clock is low, then output the fact
that the
                            // data line is not idle and reset clock count
                            begin
                                state <= LINE_NOT_IDLE;

                                clock_high_timer <= 0;
                                serial_line_idle <= 0;

                                if(serial_send_data)
                                    serial_send_failure <= 1;
                            end

                    LINE_IDLE:
                        if(!ps2_clock_in)
                            // If was idle, but clock is now low, then the
data line
                            // is no longer idle.  Return to non-idle
state and output
                            // fact that line is no longer idle
                            begin
                                state <= LINE_NOT_IDLE;
```

```
                            clock_high_timer <= 0;
                            serial_line_idle <= 0;
                        end
                    else if(serial_send_data)
                        // If line is idle and receive byte of data to
send, then
                        // start sending data.
                        begin
                            // Enter sending, high clock state.  This
assumes that
                            // the clock generator will start off with
a high
                            // clock signal
                            state <= SENDING_PS2_CLOCK_HIGH;
                            clock_high_timer <= 1;

                            // Enable clock generator
                            clock_gen_clock_enable <= 1;

                            // Store new packet into shift registers.
                            // Calculate odd parity bit and surround
packet with
                            // start and stop bits
                            ps2_data_out <= 1;
                            num_bits_sent <= 0;
                            data_to_send_shift_regs[0] <= 0;    //
Start bit
                            data_to_send_shift_regs[8:1] <=
serial_data_to_send[7:0];    // Data bits
                            data_to_send_shift_regs[9] <=       //
Parity bit
                                ~(((serial_data_to_send[7] ^
serial_data_to_send[6]) ^
                                   (serial_data_to_send[5] ^
serial_data_to_send[4])) ^
                                  ((serial_data_to_send[3] ^
serial_data_to_send[2]) ^
                                   (serial_data_to_send[1] ^
serial_data_to_send[0])));
                            data_to_send_shift_regs[10] <= 1;   // Stop
bit

                            // Line is not idle since busy sending data
                            serial_line_idle <= 0;
                        end
                    else
                        // Line is idle, did not receive data to send.
Continue
                        // in idle state
                        begin
                            state <= LINE_IDLE;

                            clock_high_timer <= 0;
                            serial_line_idle <= 1;       // Output that
line is idle
                        end
```

```
                    SENDING_PS2_CLOCK_HIGH:
                        if(!ps2_clock_in)
                            // If PS/2 clock was high, but is now low,
enter sending,
                            // low clock state.
                            begin
                                state <= SENDING_PS2_CLOCK_LOW;
                                clock_high_timer <= 0;

                                // Keep clock generator on
                                clock_gen_clock_enable <= 1;

                                // Maintain data bit that is presently on
the PS/2 data
                                // line.  Maintain the data to send and the
number of
                                // bits already sent.
                                ps2_data_out <= ps2_data_out;
                                num_bits_sent <= num_bits_sent;
                                data_to_send_shift_regs[10:0] <=
data_to_send_shift_regs[10:0];

                                // Line is not idle since busy sending data
                                serial_line_idle <= 0;
                            end
                        else if(clock_high_timer <
NUM_CYCLES_CLOCK_HIGH_BEFORE_SEND - 1)
                            // Count time until time for data transition
during the
                            // high period of the PS/2 clock signal
                            begin
                                state <= SENDING_PS2_CLOCK_HIGH;
                                clock_high_timer <= clock_high_timer + 1;

                                // Keep clock generator on
                                clock_gen_clock_enable <= 1;

                                // Maintain data bit that is presently on
the PS/2 data
                                // line.  Maintain the data to send and the
number of
                                // bits already sent.
                                ps2_data_out <= ps2_data_out;
                                num_bits_sent <= num_bits_sent;
                                data_to_send_shift_regs[10:0] <=
data_to_send_shift_regs[10:0];

                                // Line is not idle since busy sending data
                                serial_line_idle <= 0;
                            end
                        else if(clock_high_timer ==
NUM_CYCLES_CLOCK_HIGH_BEFORE_SEND - 1)
                            // If time for data transition, shift out new
bit onto the
                            // PS/2 data line
                            begin
```

```
                            state <= SENDING_PS2_CLOCK_HIGH;
                            clock_high_timer <= clock_high_timer + 1;

                            // Keep clock generator on
                            clock_gen_clock_enable <= 1;

                            // Shift new data bit onto PS/2 data line.
Increment
                            // the number of bits sent and shift bits
in shift
                            // registers
                            ps2_data_out <= data_to_send_shift_regs[0];
                            num_bits_sent <= num_bits_sent + 1;
                            data_to_send_shift_regs[0] <=
data_to_send_shift_regs[1];
                            data_to_send_shift_regs[1] <=
data_to_send_shift_regs[2];
                            data_to_send_shift_regs[2] <=
data_to_send_shift_regs[3];
                            data_to_send_shift_regs[3] <=
data_to_send_shift_regs[4];
                            data_to_send_shift_regs[4] <=
data_to_send_shift_regs[5];
                            data_to_send_shift_regs[5] <=
data_to_send_shift_regs[6];
                            data_to_send_shift_regs[6] <=
data_to_send_shift_regs[7];
                            data_to_send_shift_regs[7] <=
data_to_send_shift_regs[8];
                            data_to_send_shift_regs[8] <=
data_to_send_shift_regs[9];
                            data_to_send_shift_regs[9] <=
data_to_send_shift_regs[10];
                            data_to_send_shift_regs[10] <= 0;

                            // Line is not idle since busy sending data
                            serial_line_idle <= 0;
                        end
                    else
                        // Time for data transition during PS/2 clock
high has
                        // already passed.  Wait until the PS/2 clock
transitions
                        // to low
                        begin
                            state <= SENDING_PS2_CLOCK_HIGH;
                            clock_high_timer <= clock_high_timer;

                            // Keep clock generator on
                            clock_gen_clock_enable <= 1;

                            // Maintain data bit that is presently on
the PS/2 data
                            // line.  Maintain the data to send and the
number of
                            // bits already sent.
                            ps2_data_out <= ps2_data_out;
```

```
                            num_bits_sent <= num_bits_sent;
                            data_to_send_shift_regs[10:0] <=
data_to_send_shift_regs[10:0];

                            // Line is not idle since busy sending data
                            serial_line_idle <= 0;
                        end

                    SENDING_PS2_CLOCK_LOW:
                        if(ps2_clock_in)
                            // If PS/2 clock was low, but is now high,
check to see if
                            // need to send another bit.  If need to send
another bit,
                            // transition to sending, clock high state.
If no more
                            // bits to send, then return to not sending,
not idle
                            // state (since idleness needs to be
reestablished)
                            if(num_bits_sent < 11)
                                begin
                                    // If need to send another bit, prepare
to do so
                                    state <= SENDING_PS2_CLOCK_HIGH;
                                    clock_high_timer <= 1;

                                    // Keep clock generator on
                                    clock_gen_clock_enable <= 1;

                                    // Maintain data bit that is presently
on the PS/2 data
                                    // line.  Maintain the data to send and
the number of
                                    // bits already sent.
                                    ps2_data_out <= ps2_data_out;
                                    num_bits_sent <= num_bits_sent;
                                    data_to_send_shift_regs[10:0] <=
data_to_send_shift_regs[10:0];

                                    // Line is not idle since busy sending
data
                                    serial_line_idle <= 0;
                                end
                            else
                                begin
                                    // No more bits to send return to
default state
                                    state <= LINE_NOT_IDLE;
                                    clock_high_timer <= 0;

                                    // Turn off clock generator, don't need
to send
                                    // anything else
                                    clock_gen_clock_enable <= 0;
```

```
                                // Set PS/2 data line to default value
of 1.  Clear
                                // shift registers used for storing
data to send and
                                // reset number of bits sent
                                ps2_data_out <= 1;
                                num_bits_sent <= 0;
                                data_to_send_shift_regs[10:0] <= 0;

                                // Line not idle, need to reestablished
idleness
                                // Sending was successful
                                serial_line_idle <= 0;
                                serial_send_success <= 1;
                            end
                        else if(ps2_inhibit)
                            // If PS/2 clock was low and host has
inhibited
                            // communication, terminate sending of current
byte.
                            // Check to see if already finished
transmitting byte.
                            // If so, return serial_send_success.
Otherwise,
                            // return serial_send_failure
                            begin
                                state <= LINE_NOT_IDLE;
                                clock_high_timer <= 0;

                                // Turn off clock generator, don't need to
send
                                // anything else
                                clock_gen_clock_enable <= 0;

                                // Set PS/2 data line to default value of
1.  Clear
                                // shift registers used for storing data to
send and
                                // reset number of bits sent
                                ps2_data_out <= 1;
                                num_bits_sent <= 0;
                                data_to_send_shift_regs[10:0] <= 0;

                                // Line not idle, need to reestablished
idleness
                                // Success/failure dependent on if byte
already
                                // finished sending
                                serial_line_idle <= 0;

                                if(num_bits_sent < 11)
                                    serial_send_failure <= 1;
                                else
                                    serial_send_success <= 1;
                            end
                        else
```

```
                        // PS/2 clock is still low.  Stay in the
sending, low
                        // clock state
                        begin
                            state <= SENDING_PS2_CLOCK_LOW;
                            clock_high_timer <= 0;

                            // Keep clock generator on
                            clock_gen_clock_enable <= 1;

                            // Maintain data bit that is presently on
the PS/2 data
                            // line.  Maintain the data to send and the
number of
                            // bits already sent.
                            ps2_data_out <= ps2_data_out;
                            num_bits_sent <= num_bits_sent;
                            data_to_send_shift_regs[10:0] <=
data_to_send_shift_regs[10:0];

                            // Line is not idle since busy sending data
                            serial_line_idle <= 0;
                        end
                endcase
        end
endmodule
```

ps2_typematic_converter.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////
////////////
//
// Ceryen Tan
//
// This module converts action enables into make and break codes that
are to be
// sent over the PS/2 interface.  This module takes in make scan codes
for each
// action and sends them when the action is enabled.  Break scan codes
// (generated from the make scan code) are sent when the action is
disabled.
//
// This module also handles typematic behavior.  When an action is
enabled long
// enough, the make code for that action starts being sent
periodically.  This
// typematic behavior is stopped when the action is disabled or when
another
// action in enabled
//
// Make and break codes are enqueued to the keyboard buffer module
//
//////////////////////////////////////////////////////////////////////
////////////
```

```
module ps2_typematic_converter(reset, clk_27_mhz,
                                       action_enable_0,
                                       action_enable_1,
                                       action_enable_2,
                                       action_enable_3,
                                       action_enable_4,
                                       action_enable_5,
                                       action_enable_6,
                                       action_enable_7,
                                       action_make_code_0,
                                       action_make_code_1,
                                       action_make_code_2,
                                       action_make_code_3,
                                       action_make_code_4,
                                       action_make_code_5,
                                       action_make_code_6,
                                       action_make_code_7,
                                       keyboard_buffer_enqueue_data,
                                       keyboard_buffer_data_to_enqueue);
    input reset;
    input clk_27_mhz;

    // Action enables
    input action_enable_0;
    input action_enable_1;
    input action_enable_2;
    input action_enable_3;
    input action_enable_4;
    input action_enable_5;
    input action_enable_6;
    input action_enable_7;

    // Make codes for each action
    input [15:0] action_make_code_0;
    input [15:0] action_make_code_1;
    input [15:0] action_make_code_2;
    input [15:0] action_make_code_3;
    input [15:0] action_make_code_4;
    input [15:0] action_make_code_5;
    input [15:0] action_make_code_6;
    input [15:0] action_make_code_7;

    // Enqueues make and break codes to the keyboard buffer
    output keyboard_buffer_enqueue_data;
    output [23:0] keyboard_buffer_data_to_enqueue;

    // Action event detectors.  When an action becomes enabled,
action_enabled
    // is set high for one clock cycle.  When an action becomes
disabled,
    // action_disabled is set high for one clock cycle
    wire [7:0] action_enabled;
    wire [7:0] action_disabled;

    ps2_key_event_detector action_0_event_detector(reset, clk_27_mhz,

action_enable_0,
```

```
action_enabled[0],
action_disabled[0]);
    ps2_key_event_detector action_1_event_detector(reset, clk_27_mhz,
action_enable_1,
action_enabled[1],
action_disabled[1]);
    ps2_key_event_detector action_2_event_detector(reset, clk_27_mhz,
action_enable_2,
action_enabled[2],
action_disabled[2]);
    ps2_key_event_detector action_3_event_detector(reset, clk_27_mhz,
action_enable_3,
action_enabled[3],
action_disabled[3]);
    ps2_key_event_detector action_4_event_detector(reset, clk_27_mhz,
action_enable_4,
action_enabled[4],
action_disabled[4]);
    ps2_key_event_detector action_5_event_detector(reset, clk_27_mhz,
action_enable_5,
action_enabled[5],
action_disabled[5]);
    ps2_key_event_detector action_6_event_detector(reset, clk_27_mhz,
action_enable_6,
action_enabled[6],
action_disabled[6]);
    ps2_key_event_detector action_7_event_detector(reset, clk_27_mhz,
action_enable_7,
```

```
action_enabled[7],

action_disabled[7]);

    // Break codes generated from the make codes passed in
    wire [23:0] action_break_code_0;
    wire [23:0] action_break_code_1;
    wire [23:0] action_break_code_2;
    wire [23:0] action_break_code_3;
    wire [23:0] action_break_code_4;
    wire [23:0] action_break_code_5;
    wire [23:0] action_break_code_6;
    wire [23:0] action_break_code_7;

    ps2_break_code_gen action_0_break_code_gen(action_make_code_0,
action_break_code_0);
    ps2_break_code_gen action_1_break_code_gen(action_make_code_1,
action_break_code_1);
    ps2_break_code_gen action_2_break_code_gen(action_make_code_2,
action_break_code_2);
    ps2_break_code_gen action_3_break_code_gen(action_make_code_3,
action_break_code_3);
    ps2_break_code_gen action_4_break_code_gen(action_make_code_4,
action_break_code_4);
    ps2_break_code_gen action_5_break_code_gen(action_make_code_5,
action_break_code_5);
    ps2_break_code_gen action_6_break_code_gen(action_make_code_6,
action_break_code_6);
    ps2_break_code_gen action_7_break_code_gen(action_make_code_7,
action_break_code_7);

    // The delay necessary for an action to become typematic, and the
frequency
    // at which the action is periodically rebroadcasted after it
becomes
    // typematic
    parameter TYPEMATIC_DELAY = 6750000;
    parameter TYPEMATIC_REPEAT = 900000;

    // States that determine whether the last action enabled is
typematic
    parameter NO_NEW_KEY_PRESSED = 0;       // Last action enabled is
now disabled, can't become typematic
    parameter NEW_KEY_PRESSED = 1;          // Last action enabled is
not yet typematic
    parameter NEW_KEY_TYPEMATIC = 2;        // Last action enabled is
typematic

    reg [1:0] state;
    reg [25:0] clock_timer;

    reg [2:0] last_action_enabled;

    // Enqueues make and break codes to the keyboard buffer
    reg keyboard_buffer_enqueue_data;
    reg [23:0] keyboard_buffer_data_to_enqueue;
```

```
    initial
        begin
            state <= NO_NEW_KEY_PRESSED;
            clock_timer <= 0;

            last_action_enabled = 0;

            keyboard_buffer_enqueue_data <= 0;
            keyboard_buffer_data_to_enqueue <= 0;
        end

    always @(posedge clk_27_mhz)
        begin
            clock_timer <= 0;

            keyboard_buffer_enqueue_data <= 0;
            keyboard_buffer_data_to_enqueue <= 0;

            last_action_enabled <= last_action_enabled;

            if(reset)
                begin
                    state <= NO_NEW_KEY_PRESSED;
                    clock_timer <= 0;

                    keyboard_buffer_enqueue_data <= 0;
                    keyboard_buffer_data_to_enqueue <= 0;

                    last_action_enabled <= 0;
                end
            else if(action_disabled[0] || action_disabled[1] ||
action_disabled[2] ||
                    action_disabled[3] || action_disabled[4] ||
action_disabled[5] ||
                    action_disabled[6] || action_disabled[7])
                begin
                    // If an action is newly disabled
                    if(state == NEW_KEY_TYPEMATIC)
                        begin
                            if((last_action_enabled == 0 &&
action_disabled[0]) ||
                               (last_action_enabled == 1 &&
action_disabled[1]) ||
                               (last_action_enabled == 2 &&
action_disabled[2]) ||
                               (last_action_enabled == 3 &&
action_disabled[3]) ||
                               (last_action_enabled == 4 &&
action_disabled[4]) ||
                               (last_action_enabled == 5 &&
action_disabled[5]) ||
                               (last_action_enabled == 6 &&
action_disabled[6]) ||
                               (last_action_enabled == 7 &&
action_disabled[7]))
                               state <= NO_NEW_KEY_PRESSED;
```

```
                        else
                            state <= NEW_KEY_PRESSED;
                    end
                else
                    state <= NO_NEW_KEY_PRESSED;

                clock_timer <= 0;

                // Enqueue break code for action to keyboard buffer
                keyboard_buffer_enqueue_data <= 1;

                if(action_disabled[0])
                    keyboard_buffer_data_to_enqueue <=
action_break_code_0;
                else if(action_disabled[1])
                    keyboard_buffer_data_to_enqueue <=
action_break_code_1;
                else if(action_disabled[2])
                    keyboard_buffer_data_to_enqueue <=
action_break_code_2;
                else if(action_disabled[3])
                    keyboard_buffer_data_to_enqueue <=
action_break_code_3;
                else if(action_disabled[4])
                    keyboard_buffer_data_to_enqueue <=
action_break_code_4;
                else if(action_disabled[5])
                    keyboard_buffer_data_to_enqueue <=
action_break_code_5;
                else if(action_disabled[6])
                    keyboard_buffer_data_to_enqueue <=
action_break_code_6;
                else if(action_disabled[7])
                    keyboard_buffer_data_to_enqueue <=
action_break_code_7;

                last_action_enabled <= last_action_enabled;
            end
        else if(action_enabled[0] || action_enabled[1] ||
action_enabled[2] ||
                action_enabled[3] || action_enabled[4] ||
action_enabled[5] ||
                action_enabled[6] || action_enabled[7])
            begin
                // If an action is newly enabled, then new key
pressed
                state <= NEW_KEY_PRESSED;
                clock_timer <= 0;

                // Enqueue the make code for the action that is newly
enabled
                // Keep track of last action that is enable
                keyboard_buffer_enqueue_data <= 1;

                if(action_enabled[0])
                    begin
                        last_action_enabled <= 0;
```

```
                        keyboard_buffer_data_to_enqueue <=
{action_make_code_0, 8'b0};
                    end
                else if(action_enabled[1])
                    begin
                        last_action_enabled <= 1;
                        keyboard_buffer_data_to_enqueue <=
{action_make_code_1, 8'b0};
                    end
                else if(action_enabled[2])
                    begin
                        last_action_enabled <= 2;
                        keyboard_buffer_data_to_enqueue <=
{action_make_code_2, 8'b0};
                    end
                else if(action_enabled[3])
                    begin
                        last_action_enabled <= 3;
                        keyboard_buffer_data_to_enqueue <=
{action_make_code_3, 8'b0};
                    end
                else if(action_enabled[4])
                    begin
                        last_action_enabled <= 4;
                        keyboard_buffer_data_to_enqueue <=
{action_make_code_4, 8'b0};
                    end
                else if(action_enabled[5])
                    begin
                        last_action_enabled <= 5;
                        keyboard_buffer_data_to_enqueue <=
{action_make_code_5, 8'b0};
                    end
                else if(action_enabled[6])
                    begin
                        last_action_enabled <= 6;
                        keyboard_buffer_data_to_enqueue <=
{action_make_code_6, 8'b0};
                    end
                else if(action_enabled[7])
                    begin
                        last_action_enabled <= 7;
                        keyboard_buffer_data_to_enqueue <=
{action_make_code_7, 8'b0};
                    end
            end
        else
            case(state)
                // Last action enabled is now disabled, can't become
typematic
                NO_NEW_KEY_PRESSED:
                    begin
                        state <= NO_NEW_KEY_PRESSED;
                        clock_timer <= 0;
                    end

                // Last action enabled is not yet typematic
```

```
                    NEW_KEY_PRESSED:
                        // Check to see if action has been active long
enough to be
                        // typematic.  If so, retransmit make code and
become
                        // typematic
                        if(clock_timer == TYPEMATIC_DELAY - 1)
                            begin
                                state <= NEW_KEY_TYPEMATIC;
                                clock_timer <= 0;

                                keyboard_buffer_enqueue_data <= 1;

                                case(last_action_enabled)
                                    0: keyboard_buffer_data_to_enqueue <=
{action_make_code_0, 8'b0};
                                    1: keyboard_buffer_data_to_enqueue <=
{action_make_code_1, 8'b0};
                                    2: keyboard_buffer_data_to_enqueue <=
{action_make_code_2, 8'b0};
                                    3: keyboard_buffer_data_to_enqueue <=
{action_make_code_3, 8'b0};
                                    4: keyboard_buffer_data_to_enqueue <=
{action_make_code_4, 8'b0};
                                    5: keyboard_buffer_data_to_enqueue <=
{action_make_code_5, 8'b0};
                                    6: keyboard_buffer_data_to_enqueue <=
{action_make_code_6, 8'b0};
                                    7: keyboard_buffer_data_to_enqueue <=
{action_make_code_7, 8'b0};
                                endcase
                            end

                        // Otherwise, count the amount of time that the
action has
                        // been active
                        else
                            begin
                                state <= NEW_KEY_PRESSED;
                                clock_timer <= clock_timer + 1;

                                keyboard_buffer_enqueue_data <= 0;
                                keyboard_buffer_data_to_enqueue <= 0;
                            end

                    // Last action enabled is typematic
                    NEW_KEY_TYPEMATIC:
                        // Check to see if time to rebroadcast action has
been reached
                        if(clock_timer == TYPEMATIC_REPEAT - 1)
                            begin
                                state <= NEW_KEY_TYPEMATIC;
                                clock_timer <= 0;

                                // Rebroadcast action
                                keyboard_buffer_enqueue_data <= 1;
```

```
                            case(last_action_enabled)
                                0: keyboard_buffer_data_to_enqueue <=
{action_make_code_0, 8'b0};
                                1: keyboard_buffer_data_to_enqueue <=
{action_make_code_1, 8'b0};
                                2: keyboard_buffer_data_to_enqueue <=
{action_make_code_2, 8'b0};
                                3: keyboard_buffer_data_to_enqueue <=
{action_make_code_3, 8'b0};
                                4: keyboard_buffer_data_to_enqueue <=
{action_make_code_4, 8'b0};
                                5: keyboard_buffer_data_to_enqueue <=
{action_make_code_5, 8'b0};
                                6: keyboard_buffer_data_to_enqueue <=
{action_make_code_6, 8'b0};
                                7: keyboard_buffer_data_to_enqueue <=
{action_make_code_7, 8'b0};
                            endcase
                        end

                    // Otherwise, count until next time to rebroadcast
                    else
                        begin
                            state <= NEW_KEY_TYPEMATIC;
                            clock_timer <= clock_timer + 1;

                            keyboard_buffer_enqueue_data <= 0;
                            keyboard_buffer_data_to_enqueue <= 0;
                        end
                endcase
        end
endmodule
```

Keymapper

fsm.v

```
`timescale 1ns / 1ps

module fsm(reset, clock, up, down, left, right, enter, chars, uv, dv,
lv, rv, av, bv);
    input clock, reset;

    // buttons from labkit
    input up, down, left, right, enter;

    // output characters
    output [127:0] chars;
    reg [127:0] chars;

    // values for each param
    output [7:0] uv;
    output [7:0] dv;
    output [7:0] lv;
    output [7:0] rv;
    output [7:0] av;
    output [7:0] bv;

    // ...and the corresponding regs
    reg [7:0] uv;
    reg [7:0] dv;
    reg [7:0] lv;
    reg [7:0] rv;
    reg [7:0] av;
    reg [7:0] bv;
    reg [7:0] dispv;

    // internal state register
    reg [3:0] state;

    // values assigned to states of fsm
    parameter UP        = 4'd0;
    parameter DOWN      = 4'd1;
    parameter LEFT      = 4'd2;
    parameter RIGHT     = 4'd3;
    parameter A         = 4'd4;
    parameter B         = 4'd5;
    parameter UP_SEL    = 4'd6;
    parameter DOWN_SEL  = 4'd7;
    parameter LEFT_SEL  = 4'd8;
    parameter RIGHT_SEL = 4'd9;
    parameter A_SEL     = 4'd10;
    parameter B_SEL     = 4'd11;

    always @(posedge clock)
        if (reset) begin
            uv <= 8'd0;
```

```
    dv <= 8'd1;
    lv <= 8'd2;
    rv <= 8'd3;
    av <= 8'd4;
    bv <= 8'd5;
    state <= UP;
end
else
    case (state)
        UP: begin
            // display 'UP'
            chars[127:112] <= 16'h1E;  // U
            chars[111:96]  <= 16'h19;  // P
            chars[95:80]   <= 16'hFF;
            chars[79:64]   <= 16'hFF;
            chars[63:48]   <= 16'hFF;
            chars[47:32]   <= 16'hFF;
            chars[31:16]   <= 16'hFF;
            chars[15:0]    <= 16'hFF;

            if (enter) begin
                state <= UP_SEL;
                dispv <= uv;
            end
            else if (right) state <= DOWN;
            else if (left)  state <= B;
        end

        DOWN: begin // down is the selected menu item
            // display 'DOWN'
            chars[127:112] <= 16'h0D;  // D
            chars[111:96]  <= 16'h18;  // O
            chars[95:80]   <= 16'h20;  // W
            chars[79:64]   <= 16'h17;  // N
            chars[63:48]   <= 16'hFF;
            chars[47:32]   <= 16'hFF;
            chars[31:16]   <= 16'hFF;
            chars[15:0]    <= 16'hFF;

            if (enter) begin
                state <= DOWN_SEL;
                dispv <= dv;
            end
            else if (right) state <= LEFT;
            else if (left)  state <= UP;
        end

        LEFT: begin // left is the selected menu item
            // display 'LEFT'
            chars[127:112] <= 16'h15; // L
            chars[111:96]  <= 16'h0E; // E
            chars[95:80]   <= 16'h0F; // F
            chars[79:64]   <= 16'h1D; // T
            chars[63:48]   <= 16'hFF;
            chars[47:32]   <= 16'hFF;
            chars[31:16]   <= 16'hFF;
            chars[15:0]    <= 16'hFF;
```

```
    if (enter) begin
        state <= LEFT_SEL;
        dispv <= lv;
    end
    else if (right) state <= RIGHT;
    else if (left)  state <= DOWN;
end

RIGHT: begin // right is the selected menu item
    // display 'RIGHT'
    chars[127:112] <= 16'h1B; // R
    chars[111:96]  <= 16'h12; // I
    chars[95:80]   <= 16'h10; // G
    chars[79:64]   <= 16'h11; // H
    chars[63:48]   <= 16'h1D; // T
    chars[47:32]   <= 16'hFF;
    chars[31:16]   <= 16'hFF;
    chars[15:0]    <= 16'hFF;

    if (enter) begin
        state <= RIGHT_SEL;
        dispv <= rv;
    end
    else if (right) state <= A;
    else if (left)  state <= LEFT;
end

A: begin // a is the selected menu item
    // display 'A'
    chars[127:112] <= 16'h0A; // A
    chars[111:96]  <= 16'hFF;
    chars[95:80]   <= 16'hFF;
    chars[79:64]   <= 16'hFF;
    chars[63:48]   <= 16'hFF;
    chars[47:32]   <= 16'hFF;
    chars[31:16]   <= 16'hFF;
    chars[15:0]    <= 16'hFF;

    if (enter) begin
        state <= A_SEL;
        dispv <= av;
    end
    else if (right) state <= B;
    else if (left)  state <= RIGHT;
end

B: begin // b is the selected menu item
    // display 'B'
    chars[127:112] <= 16'h0B; // B
    chars[111:96]  <= 16'hFF;
    chars[95:80]   <= 16'hFF;
    chars[79:64]   <= 16'hFF;
    chars[63:48]   <= 16'hFF;
    chars[47:32]   <= 16'hFF;
    chars[31:16]   <= 16'hFF;
    chars[15:0]    <= 16'hFF;
```

```
                if (enter) begin
                    state <= B_SEL;
                    dispv <= bv;
                end
                else if (right) state <= UP;
                else if (left)  state <= A;
            end

            UP_SEL: begin // changing up value
                chars [7:0] <= dispv; // display the current
selection

                if (enter) begin
                    state <= UP;
                    uv <= dispv;
                end
                else if (up && dispv < 8'h23)  dispv <= dispv + 8'h1;
// ceiling the increase
                else if (down && dispv > 8'h0) dispv <= dispv - 8'h1;
// floor the decrease
            end

            DOWN_SEL: begin // changing down value
                chars [7:0] <= dispv; // display the current
selection

                if (enter) begin
                    state <= DOWN;
                    dv <= dispv;
                end
                else if (up && dispv < 8'h23)  dispv <= dispv + 8'h1;
// ceiling the increase
                else if (down && dispv > 8'h0) dispv <= dispv - 8'h1;
// floor the decrease
            end

            LEFT_SEL: begin // changing left value
                chars [7:0] <= dispv; // display the current
selection

                if (enter) begin
                    state <= LEFT;
                    lv <= dispv;
                end
                else if (up && dispv < 8'h23)  dispv <= dispv + 8'h1;
// ceiling the increase
                else if (down && dispv > 8'h0) dispv <= dispv - 8'h1;
// floor the decrease
            end

            RIGHT_SEL: begin // changing right value
                chars [7:0] <= dispv; // display the current
selection

                if (enter) begin
                    state <= RIGHT;
```

```
                    rv <= dispv;
                end
                else if (up && dispv < 8'h23)  dispv <= dispv + 8'h1;
// ceiling the increase
                else if (down && dispv > 8'h0) dispv <= dispv - 8'h1;
// floor the decrease
            end

            A_SEL: begin // changing a value
                chars [7:0] <= dispv; // display the current
selection

                if (enter) begin
                    state <= A;
                    av <= dispv;
                end
                else if (up && dispv < 8'h23)  dispv <= dispv + 8'h1;
// ceiling the increase
                else if (down && dispv > 8'h0) dispv <= dispv - 8'h1;
// floor the decrease
            end

            B_SEL: begin // changing b value
                chars [7:0] <= dispv; // display the current
selection

                if (enter) begin
                    state <= B;
                    bv <= dispv;
                end
                else if (up && dispv < 8'h23)  dispv <= dispv + 8'h1;
// ceiling the increase
                else if (down && dispv > 8'h0) dispv <= dispv - 8'h1;
// floor the decrease
            end

            default: begin
            end

        endcase



endmodule
```

keymapper.v

```
`timescale 1ns / 1ps

module keymapper(reset, clock, up, down, left, right, enter,
                 up_sc, down_sc, left_sc, right_sc, a_sc, b_sc, chars);

    input reset, clock;
    input up, down, left, right, enter; // button signals from labkit
```

```
    output up_sc, down_sc, left_sc, right_sc, a_sc, b_sc; // scancodes
for each button
    output [127:0] chars; // to dots modules

    wire up_d, down_d, left_d, right_d, enter_d; // debounced button
signals
    wire up_d_p, down_d_p, left_d_p, right_d_p, enter_d_p; // pulsed and
debounced button signals
    wire [7:0] uv, dv, lv, rv, av, bv; // values to be turned into
scancodes
    wire [15:0] usc, dsc, lsc, rsc, asc, bsc; // scancodes for output
    wire [127:0] chars; // to dots modules

    // debounce the button signals
    debounce debounce_u(reset, clock, button_up, up_d);
    debounce debounce_d(reset, clock, button_down, down_d);
    debounce debounce_l(reset, clock, button_left, left_d);
    debounce debounce_r(reset, clock, button_right, right_d);
    debounce debounce_e(reset, clock, button_enter, enter_d);

    // convert debounced button signals to pulses
    pulse pulse_u(reset, clock, up_d, up_d_p);
    pulse pulse_d(reset, clock, down_d, down_d_p);
    pulse pulse_l(reset, clock, left_d, left_d_p);
    pulse pulse_r(reset, clock, right_d, right_d_p);
    pulse pulse_e(reset, clock, enter_d, enter_d_p);

    // the fsm in charge
    fsm fsm1(reset, clock_27mhz, up_d_p, down_d_p, left_d_p, right_d_p,
enter_d_p, chars,
             uv, dv, lv, rv, av, bv);

    // convert stored values for actions to scancodes
    scancodes scancodes_u(clock_27mhz, uv, usc);
    scancodes scancodes_d(clock_27mhz, dv, dsc);
    scancodes scancodes_l(clock_27mhz, lv, lsc);
    scancodes scancodes_r(clock_27mhz, rv, rsc);
    scancodes scancodes_a(clock_27mhz, av, asc);
    scancodes scancodes_b(clock_27mhz, bv, bsc);

endmodule
```

pulse.v

```
`timescale 1ns / 1ps

module pulse(reset, clock, in, out);

    input clock, reset;
    input in;
    output out;

    reg high_last;
```

```
    reg out;

    always @(posedge clock) begin
        out <= 0; // keep out from being high for more than one cycle
        if (reset)
            high_last <= 0;
        else
            if (in && !high_last) begin // input is high, but was low
last cycle
                high_last <= 1;
                out <= 1;
            end
            else if (!in && high_last) //  input is low, but was high
last cycle
                high_last <= 0;
    end


endmodule
```

scancodes.v

```
`timescale 1ns / 1ps

module scancodes(clock, num, scancode);

    input clock;
    input [7:0] num;
    output [15:0] scancode;

    reg [15:0] scancode;

    always @(posedge clock)
        case (num)
            8'h00: scancode <= 16'h4500; // 0
            8'h01: scancode <= 16'h1600; // 1
            8'h02: scancode <= 16'h1E00; // 2
            8'h03: scancode <= 16'h2600; // 3
            8'h04: scancode <= 16'h2500; // 4
            8'h05: scancode <= 16'h2E00; // 5
            8'h06: scancode <= 16'h3600; // 6
            8'h07: scancode <= 16'h3D00; // 7
            8'h08: scancode <= 16'h3E00; // 8
            8'h09: scancode <= 16'h4600; // 9
            8'h0A: scancode <= 16'h1C00; // A
            8'h0B: scancode <= 16'h3200; // B
            8'h0C: scancode <= 16'h2100; // C
            8'h0D: scancode <= 16'h2300; // D
            8'h0E: scancode <= 16'h2400; // E
            8'h0F: scancode <= 16'h2B00; // F
            8'h10: scancode <= 16'h3400; // G
            8'h11: scancode <= 16'h3300; // H
            8'h12: scancode <= 16'h4300; // I
```

```
            8'h13: scancode <= 16'h3B00; // J
            8'h14: scancode <= 16'h4200; // K
            8'h15: scancode <= 16'h4B00; // L
            8'h16: scancode <= 16'h3A00; // M
            8'h17: scancode <= 16'h3100; // N
            8'h18: scancode <= 16'h4400; // O
            8'h19: scancode <= 16'h4D00; // P
            8'h1A: scancode <= 16'h1500; // Q
            8'h1B: scancode <= 16'h2D00; // R
            8'h1C: scancode <= 16'h1B00; // S
            8'h1D: scancode <= 16'h2C00; // T
            8'h1E: scancode <= 16'h3C00; // U
            8'h1F: scancode <= 16'h2A00; // V
            8'h20: scancode <= 16'h1D00; // W
            8'h21: scancode <= 16'h2200; // X
            8'h22: scancode <= 16'h3500; // Y
            8'h23: scancode <= 16'h1A00; // Z
            default: scancode <= 16'h0000;
        endcase

endmodule
```