

Appendix

alphanumeric_display.v

```
////////////////////////////////////
////////
//
// 6.111 FPGA Labkit -- Alphanumeric Display Interface for Lab 3 Memory
Tester
//
//
// Created: November 5, 2003
// Author: Nathan Ickes
// Updated by Chris Buenrostro - 5/01/2006
// Updated by Mike Scharfstein - 1/30/2006
//
// Notes:
//
// *** Code written for Agilent Technologies HCMS-2973
Alphanumeric Displays
//
// 1) manual_reset:
// a) when low, systems runs as initialized
from power up
// b) when high, system reloads control words
//
// 2) disp_test:
// a) when low, system loads dots (module input)
into dot data register
// b) when high, system loads 640 1's into dot
data register
//
// 3) disp_blank:
// a) always configured low for zero blanking
//
// 4) disp_clock:
// a) set at 1/54th global_clock speed ( .5Mhz
for 27Mhz input clock)
//
// 5) control_reg_value (parameter):
// a) controls brightness of LED banks (four
banks)
// 1) values for desired brightness can be
found in datasheet
//
////////////////////////////////////
////////

module alphanumeric_displays
(
    global_clock, manual_reset, disp_test,
```

```

disp_blank, disp_clock, disp_rs, disp_ce_b, disp_reset_b,
disp_data_out,
dots
);

input          global_clock;
input          manual_reset;
output         disp_blank;
output         disp_clock;
output         disp_data_out;          // serial
data to displays
output         disp_rs;                // register
select
output         disp_ce_b;              // chip
enable
output         disp_reset_b;          // display
reset
input [639:0]  dots;
input          disp_test;              //
check functionality of display

assign         disp_blank = 1'b0;      // 0=not
blanked

reg            disp_clock;
reg            disp_data_out;
reg            disp_rs;
reg            disp_ce_b;
reg            disp_reset_b;
// Internal signals
reg [5:0]      count;
reg [7:0]      state;
reg [9:0]      dot_index;
reg [639:0]    ldots;
////////////////////
//
//          //      There are four Control Words
//          //
//          //
//          //      parameter
control_reg_load_length = 8; //first four Banks
//          //      parameter
control_reg_load_length = 16; //second four Banks
//          //      parameter
control_reg_load_length = 24; //third four Banks
//          //      parameter control_reg_load_length
= 32;          //all four Banks
reg [control_reg_load_length - 1:0] control;

parameter     RESET
              =      0;
parameter     END_RESET
              =      1;
parameter     INIT_DOT_REG
              =      2;
parameter     LATCH_INIT_DOT_DATA
              =      3;

```

```

parameter
    SELECT_CONTROL_REG      =      4;
parameter
    =      5;
parameter
    =      6;
parameter
    =      7;
parameter
    =      8;
parameter
    =      9;
parameter
    =      10;

parameter
    = 32'h7F7F7F7F; // Controls LED brightness

////////////////////////////////////////////////////////////////
/////
//
// Initial Reset Generation
//
//     *** SRL16 is a variable-width shift register
//
////////////////////////////////////////////////////////////////
/////

wire
    reset;

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

////////////////////////////////////////////////////////////////
/////
//
// Sequential Logic Block
//
////////////////////////////////////////////////////////////////
/////

always @(posedge global_clock)
begin
    if (reset || manual_reset)
        begin
            count
                <= 0;
            disp_clock
                <= 0;
            disp_data_out
                <= 0;
            disp_rs
                <= 0;
            disp_ce_b
                <= 1;
            disp_reset_b
                <= 0;
        end
    end

```

```

        dot_index      <= 0;
        state          <= 0;
        control        <= control_reg_value;
    end
else if (count==26)
    begin
        count          <= count+1;
        disp_clock     <= 1;
    end
else if (count==53)
    begin
        count          <= 0;
        disp_clock     <= 0;

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

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

        casex (state)

            RESET:
            begin
                disp_data_out      <= 1'b0;
                disp_rs             <= 1'b0; // dot
register
                disp_ce_b          <= 1'b1;
                disp_reset_b       <= 1'b0;
                dot_index           <= 0;
                state               <= END_RESET;
            end
            END_RESET:
            begin
                disp_reset_b       <= 1'b1;
                state               <= INIT_DOT_REG;
            end
            INIT_DOT_REG:
            begin
                disp_ce_b          <= 1'b0;
                disp_data_out      <= 1'b0;

                if (dot_index == 0)
                    state          <= LATCH_INIT_DOT_DATA;
                else
                    dot_index      <= dot_index-1;
                end
            LATCH_INIT_DOT_DATA:
            begin
                disp_ce_b          <= 1'b1;
                dot_index          <= control_reg_load_length-1;
                state               <= SELECT_CONTROL_REG;
            end
end

```

```

SELECT_CONTROL_REG:
begin
    disp_rs                <= 1'b1;
    state                  <= SET_CONTROL_REG;
end
SET_CONTROL_REG:
begin
    disp_ce_b <= 1'b0;
    disp_data_out                <= control[dot_index];

    if (dot_index == 0)
        state                  <= LATCH_CONTROL_REG;
    else
        dot_index              <=
dot_index-1;
end
LATCH_CONTROL_REG:
begin
    disp_ce_b                <= 1'b1;
    dot_index                <= 639;
    state                    <= SELECT_DOT_REG;
end
SELECT_DOT_REG:
begin
    disp_rs                <= 1'b0;
    state                  <= FILL_DOT_REG;
end
FILL_DOT_REG:
begin
    disp_ce_b                <= 1'b0;

    if (disp_test)
        disp_data_out        <= 1'b1;
    else
        disp_data_out        <= ldots[dot_index];

    if (dot_index == 0)
        state                  <= LATCH_DOT_DATA;
    else
        dot_index              <= dot_index-1;
end
LATCH_DOT_DATA:
begin
    disp_ce_b                <= 1'b1;
    dot_index                <= 639;
    state                    <= LOAD_NEW_DOT_DATA;
end
LOAD_NEW_DOT_DATA:
begin
    ldots                    <= dots;
    state                    <= FILL_DOT_REG;
end
default: state                <= state;
endcase
end
else
    count <= count+1;

```

```
end  
endmodule
```

audio.v

```
module audio (reset, clock_27mhz, audio_reset_b, ac97_sdata_out,  
ac97_sdata_in,  
ac97_synch, ac97_bit_clock);  
  
input reset, clock_27mhz;  
output audio_reset_b;  
output ac97_sdata_out;  
input ac97_sdata_in;  
output ac97_synch;  
input ac97_bit_clock;  
  
wire ready;  
wire [7:0] command_address;  
wire [15:0] command_data;  
wire command_valid;  
// AC'97 spec requires 20 bit values for slots.  
  
// But the LM4550 only uses the 18 most significant bits.  
  
reg [19:0] left_out_data, right_out_data;  
wire [19:0] left_in_data, right_in_data;  
  
//  
// Reset controller. This requires an external clock such as  
clock_27mhz  
//  
reg audio_reset_b;  
reg [9:0] reset_count;  
  
always @(posedge clock_27mhz) begin  
if (reset)  
begin  
audio_reset_b = 1'b0;  
reset_count = 0;  
end  
else if (reset_count == 1023)  
audio_reset_b = 1'b1;  
else  
reset_count = reset_count+1;  
end  
  
ac97 ac97(ready, command_address, command_data, command_valid,  
left_out_data, 1'b1, right_out_data, 1'b1, left_in_data,  
right_in_data, ac97_sdata_out, ac97_sdata_in, ac97_synch,  
ac97_bit_clock);  
  
ac97commands cmds(clock_27mhz, ready, command_address, command_data,
```

```
        command_valid);

    always @(left_in_data or right_in_data)
        begin

            // digital loopback. 20 bit sample values declared
above.
            left_out_data = left_in_data;
            right_out_data = right_in_data;
        end
    endmodule

////////////////////////////////////
//////////
/*

    Finite state machine which loops through an audio frame based on
    the bit_clock.

    During this loop it serializes values for digital to analog
    conversion and

    constructs a parallel value from the serialized analog to digital
    conversion.

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

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

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

    reg ready;

    reg ac97_sdata_out;
    reg ac97_synch;

    reg [7:0] bit_count;

    reg [19:0] l_cmd_addr;
    reg [19:0] l_cmd_data;
    reg [19:0] l_left_data, l_right_data;
```

```
reg l_cmd_v, l_left_v, l_right_v;
reg [19:0] left_in_data, right_in_data;

/*
    Evil trick to initialize for simulation via initial block
    command and FPGA via specialized comments.
*/
initial begin
    ready <= 1'b0;
    // synthesis attribute init of ready is "0";
    ac97_sdata_out <= 1'b0;
    // synthesis attribute init of ac97_sdata_out is "0";
    ac97_synch <= 1'b0;
    // synthesis attribute init of ac97_synch is "0";

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

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

// Construct a frame bit by bit. Note parallel to serial
conversion.
always @(posedge ac97_bit_clock) begin
    // Generate the sync signal
    if (bit_count == 255)
        ac97_synch <= 1'b1;
    if (bit_count == 15)
        ac97_synch <= 1'b0;

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

    // Latch user data at the end of each frame. This ensures that
the
    // first frame after reset will be empty.
    if (bit_count == 255)
    begin
        l_cmd_addr <= {command_address, 12'h000};
        l_cmd_data <= {command_data, 4'h0};
    end
end
```

```

        l_cmd_v <= command_valid;
        l_left_data <= left_data;
        l_left_v <= left_valid;
        l_right_data <= right_data;
        l_right_v <= right_valid;
    end

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

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

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

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

        bit_count <= bit_count+1;

    end // always @ (posedge ac97_bit_clock)

    // Construct a sample bit by bit. Note serial to parallel
    conversion.
    always @(negedge ac97_bit_clock) begin
        if ((bit_count >= 57) && (bit_count <= 76))
            // Slot 3: Left channel
            left_in_data <= { left_in_data[18:0], ac97_sdata_in };
        else if ((bit_count >= 77) && (bit_count <= 96))
            // Slot 4: Right channel
            right_in_data <= { right_in_data[18:0], ac97_sdata_in };
        end
    end

endmodule

```

```
////////////////////////////////////  
////////////////////////////////////  
  
/*  
  
    Finite state machine which continuously loops through all of the  
    commands  
  
    for configuring AC97 the audio controller. Note that volume and  
    record  
  
    source are hardwired to maximum volume and line_in respectively  
    for  
  
    brevity. These could be configured with switches from the labkit.  
  
*/  
module ac97commands (clock, ready, command_address, command_data,  
                    command_valid);  
  
    input clock;  
    input ready;  
    output [7:0] command_address;  
    output [15:0] command_data;  
    output command_valid;  
  
    reg [23:0] command;  
    reg command_valid;  
  
    reg old_ready;  
    reg done;  
    reg [3:0] state;  
  
    initial begin  
        command <= 4'h0;  
        // synthesis attribute init of command is "0";  
        command_valid <= 1'b0;  
        // synthesis attribute init of command_valid is "0";  
        done <= 1'b0;  
        // synthesis attribute init of done is "0";  
        old_ready <= 1'b0;  
        // synthesis attribute init of old_ready is "0";  
        state <= 16'h0000;  
        // synthesis attribute init of state is "0000";  
    end  
  
    assign command_address = command[23:16];  
    assign command_data = command[15:0];  
  
    wire [4:0] vol;  
    assign vol = 5'd0; // maximum volume (lowest attenuation)  
  
    always @(posedge clock) begin  
        if (ready && (!old_ready))  
            state <= state+1;  
    end  
endmodule
```

```
case (state)
4'h0: // Read ID
begin
    command <= 24'h80_0000;
    command_valid <= 1'b1;
end
    4'h1: // Read ID
    command <= 24'h80_0000;
4'h2: // Master volume
    command <= { 8'h02, 3'b000, vol, 3'b000, vol };
4'h3: // Aux volume
    command <= { 8'h04, 3'b000, vol, 3'b000, vol };
4'h4: // Mono volume
    command <= 24'h06_8000;
4'h5: // PCM volume
    command <= 24'h18_0808;
4'h6: // Record source select
    command <= 24'h1A_0404; // line-in
4'h7: // Record gain
    command <= 24'h1C_0000;
4'h8: // Line in gain
    command <= 24'h10_8000;
//4'h9: // Set jack sense pins
//command <= 24'h72_3F00;
4'hA: // Set beep volume
    command <= 24'h0A_0000;
//4'hF: // Misc control bits
//command <= 24'h76_8000;
default:
    command <= 24'h80_0000;
endcase // case(state)

old_ready <= ready;

end // always @ (posedge clock)

endmodule // ac97commands
```

audio_fft.v

```
/*
*****
* This file is owned and controlled by Xilinx and must be used
* solely for design, simulation, implementation and creation of
* design files limited to Xilinx devices or technologies. Use
* with non-Xilinx devices or technologies is expressly prohibited
* and immediately terminates your license.
*
* XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
*
*****
*/
```

```
* SOLELY FOR USE IN DEVELOPING PROGRAMS AND SOLUTIONS FOR
*
* XILINX DEVICES. BY PROVIDING THIS DESIGN, CODE, OR INFORMATION
*
* AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE, APPLICATION
*
* OR STANDARD, XILINX IS MAKING NO REPRESENTATION THAT THIS
*
* IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
*
* AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
*
* FOR YOUR IMPLEMENTATION. XILINX EXPRESSLY DISCLAIMS ANY
*
* WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
*
* IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
*
* REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
*
* INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
*
* FOR A PARTICULAR PURPOSE.
```

```
*
*
* Xilinx products are not intended for use in life support
*
* appliances, devices, or systems. Use in such applications are
*
* expressly prohibited.
```

```
* (c) Copyright 1995-2004 Xilinx, Inc.
```

```
*
* *****
* *****/
```

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

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

```
`timescale 1ns/1ps
```

```
module audio_fft(
    xn_re,
    xn_im,
    start,
```

```
    fwd_inv,  
    fwd_inv_we,  
    clk,  
    xk_re,  
    xk_im,  
    xn_index,  
    xk_index,  
    rfd,  
    busy,  
    dv,  
    edone,  
    done);  
  
input [7 : 0] xn_re;  
input [7 : 0] xn_im;  
input start;  
input fwd_inv;  
input fwd_inv_we;  
input clk;  
output [18 : 0] xk_re;  
output [18 : 0] xk_im;  
output [9 : 0] xn_index;  
output [9 : 0] xk_index;  
output rfd;  
output busy;  
output dv;  
output edone;  
output done;  
  
// synopsys translate_off  
  
    XFFT_V3_1 #(  
        3,    // c_arch  
        1,    // c_bram_stages  
        1,    // c_data_mem_type  
        0,    // c_enable_rlocs  
        "virtex2", // c_family  
        0,    // c_has_bfp  
        1,    // c_has_bypass  
        0,    // c_has_ce  
        1,    // c_has_natural_output  
        0,    // c_has_nfft  
        0,    // c_has_ovflo  
        0,    // c_has_rounding  
        0,    // c_has_scaling  
        0,    // c_has_sclr  
        8,    // c_input_width  
        10,   // c_nfft_max  
        0,    // c_optimize  
        19,   // c_output_width  
        1,    // c_twiddle_mem_type  
        8)    // c_twiddle_width  
inst (  
    .XN_RE(xn_re),  
    .XN_IM(xn_im),  
    .START(start),
```

```
.FWD_INV(fwd_inv),
.FWD_INV_WE(fwd_inv_we),
.CLK(clk),
.XK_RE(xk_re),
.XK_IM(xk_im),
.XN_INDEX(xn_index),
.XK_INDEX(xk_index),
.RFD(rfd),
.BUSY(busy),
.DV(dv),
.EDONE(edone),
.DONE(done),
.UNLOAD(),
.NFFT(),
.NFFT_WE(),
.SCALE_SCH(),
.SCALE_SCH_WE(),
.SCLR(),
.CE(),
.BLK_EXP(),
.OVFLO());

// synopsys translate_on

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

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

endmodule

audio_processor.v
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////
// Module audio_processor - Processes streaming audio data and outputs
FFT data
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////
module audio_processor(clk, ac97_data, ready, slowBars, xk_index,
xk_re, xk_im, event_enable);

    parameter [2:0] STATE_IDLE = 3'd0;
    parameter [2:0] STATE_SEND = 3'd1;
    parameter [2:0] STATE_RECV = 3'd2;

    input clk;
    input [7:0] ac97_data;
    input ready;

    output [127:0] slowBars;
```

```
// Debug outputs
output [9:0] xk_index;
output [18:0] xk_re;
output [18:0] xk_im;
output event_enable;

wire [18:0] xk_re;
wire [18:0] xk_im;
wire [9:0] xn_index;
wire [9:0] xk_index;
wire rfd;
wire busy;
wire dv;
wire edone;
wire done;
wire [7:0] xn_re;
wire [7:0] xn_im;

reg old_ready = 0;
reg [7:0] ac97_data_valid = 0;
reg [37:0] raw_data [15:0];
wire [37:0] clean_sum;
wire [127:0] bars;
reg [127:0] slow_bars;
reg [31:0] count;

assign xn_re = ac97_data_valid;
assign xn_im = 8'd0;

reg event_enable = 0;
reg [31:0] enable_count = 0;

reg [2:0] state = STATE_IDLE;
reg [2:0] next = STATE_IDLE;

assign bars[7:0] = raw_data[0][30:23];
assign bars[15:8] = raw_data[1][33:26];
assign bars[23:16] = raw_data[2][30:23];
assign bars[31:24] = raw_data[3][27:20];
assign bars[39:32] = raw_data[4][27:20];
assign bars[47:40] = raw_data[5][27:20];
assign bars[55:48] = raw_data[6][27:20];
assign bars[63:56] = raw_data[7][27:20];
assign bars[71:64] = raw_data[8][27:20];
assign bars[79:72] = raw_data[9][27:20];
assign bars[87:80] = raw_data[10][27:20];
assign bars[95:88] = raw_data[11][27:20];
assign bars[103:96] = raw_data[12][27:20];
assign bars[111:104] = raw_data[13][27:20];
assign bars[119:112] = raw_data[14][27:20];
assign bars[127:120] = raw_data[15][27:20];

audio_fft A_FFT (
    xn_re,          // Real component of input data
    xn_im,          // Imaginary component of input data
    1'd1,          // FFT start signal
```

```

        1'd1,                // Forward enable (1 = forward
transform)
        1'd0,                // Forward inv write enable (active
high)
        clk,                 // FFT Clock
        xk_re,               // Real component of output data
        xk_im,               // Imaginary component of output data
        xn_index,           // Index of input data
        xk_index,           // Index of output data
        rfd,                 // Ready for data (active high during
load)
        busy,                // Busy (high during fft computation)
        dv,                  // Data valid (high when data is valid)
        edone,               // Early done (high one cycle before
DONE)
        done);              // Done strobe (high for one cycle
after transform complete)

    sum_of_squares SOS (
        clk,
        xk_re,
        xk_im,
        clean_sum
    );

    always @(posedge clk)
    begin

        count <= count + 1;
        if (count > 31'd10000) begin
            count <= 0;
            slow_bars[7:0] <= bars[7:0];
            slow_bars[15:8] <= bars[15:8];
            slow_bars[23:16] <= bars[23:16];
            slow_bars[31:24] <= bars[31:24];
            slow_bars[39:32] <= bars[39:32];
            slow_bars[47:40] <= bars[47:40];
            slow_bars[55:48] <= bars[55:48];
            slow_bars[63:56] <= bars[63:56];
            slow_bars[71:64] <= bars[71:64];
            slow_bars[79:72] <= bars[79:72];
            slow_bars[87:80] <= bars[87:80];
            slow_bars[95:88] <= bars[95:88];
            slow_bars[103:96] <= bars[103:96];
            slow_bars[111:104] <= bars[111:104];
            slow_bars[119:112] <= bars[119:112];
            slow_bars[127:120] <= bars[127:120];
        end

        old_ready <= ready;
        if (ready && !old_ready) begin
            ac97_data_valid <= ac97_data;
        end

        if (enable_count > 0 && enable_count < 9600) begin
            enable_count <= enable_count + 1;
        end
    end

```

```
else begin
    event_enable <= 0;
    enable_count <= 0;
end

if (dv) begin
    case (xk_index)
        // Extract magnitude of the lowest frequency
        0: begin
            raw_data[0] <= clean_sum;
            if (slow_bars[15:8] > 8'd100 &&
enable_count == 0) begin
                enable_count <= 1;
                event_enable <= 1;
            end
        end
        1: raw_data[1] <= clean_sum;
        2: raw_data[2] <= clean_sum;
        3: raw_data[3] <= clean_sum;
        4: raw_data[4] <= clean_sum;
        5: raw_data[5] <= clean_sum;
        6: raw_data[6] <= clean_sum;
        7: raw_data[7] <= clean_sum;
        8: raw_data[8] <= clean_sum;
        9: raw_data[9] <= clean_sum;
        10: raw_data[10] <= clean_sum;
        11: raw_data[11] <= clean_sum;
        12: raw_data[12] <= clean_sum;
        13: raw_data[13] <= clean_sum;
        14: raw_data[14] <= clean_sum;
        15: raw_data[15] <= clean_sum;
    endcase

    //if (xk_index > 15 && xk_index < 32) begin
    //    raw_data[xk_index - 16] <= clean_sum;
//((xk_re*xk_re + xk_im*xk_im);
//end

end

end

endmodule
```

binary_to_bcd.v

```
//-----
//
// Binary to BCD converter, serial implementation, 1 clock per input
// bit.
//
// Description: See description below (which suffices for IP core
// specification document.)
//
// Copyright (C) 2002 John Clayton and OPENCORES.ORG (this Verilog
// version)
```

```
//
// This source file may be used and distributed without restriction
provided
// that this copyright statement is not removed from the file and that
any
// derivative work contains the original copyright notice and the
associated
// disclaimer.
//
// This source file is free software; you can redistribute it and/or
modify
// it under the terms of the GNU Lesser General Public License as
published
// by the Free Software Foundation; either version 2.1 of the License,
or
// (at your option) any later version.
//
// This source is distributed in the hope that it will be useful, but
WITHOUT
// ANY WARRANTY; without even the implied warranty of MERCHANTABILITY
or
// FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public
// License for more details.
//
// You should have received a copy of the GNU Lesser General Public
License
// along with this source.
// If not, download it from http://www.opencores.org/lgpl.shtml
//
//-----
-----
//
// Author: John Clayton
// Date : Nov. 19, 2003
// Update: Nov. 19, 2003 Copied this file from "led_display_driver.v"
and
// modified it.
// Update: Nov. 24, 2003 Fixed bcd_asl function, tested module. It
works!
// Update: Nov. 25, 2003 Changed bit_counter and related logic so that
long
// start pulses produce correct results at the
end of
// the pulse.
//
//-----
-----
// Description:
//
// This module takes a binary input, and converts it into BCD output,
with each
// binary coded decimal digit of course occupying 4-bits.
// The user can specify the number of input bits separately from the
number of
// output digits. Be sure that you have specified enough output digits
to
```

```

// represent the largest number you expect on the binary input, or else
the
// most significant digits of the result will be cut off.
//
//-----
-----

module binary_to_bcd (
    clk_i,
    ce_i,
    rst_i,
    start_i,
    dat_binary_i,
    dat_bcd_o,
    done_o
);
parameter BITS_IN_PP          = 13; // # of bits of binary input
parameter BCD_DIGITS_OUT_PP  = 4;  // # of digits of BCD output
parameter BIT_COUNT_WIDTH_PP = 4;  // Width of bit counter

// I/O declarations
input  clk_i;           // clock signal
input  ce_i;           // clock enable input
input  rst_i;          // synchronous reset
input  start_i;        // initiates a conversion
input  [BITS_IN_PP-1:0] dat_binary_i; // input bus
output [4*BCD_DIGITS_OUT_PP-1:0] dat_bcd_o; // output bus
output done_o;         // indicates conversion is done

reg [4*BCD_DIGITS_OUT_PP-1:0] dat_bcd_o;

// Internal signal declarations

reg [BITS_IN_PP-1:0] bin_reg;
reg [4*BCD_DIGITS_OUT_PP-1:0] bcd_reg;
wire [BITS_IN_PP-1:0] bin_next;
reg [4*BCD_DIGITS_OUT_PP-1:0] bcd_next;
reg busy_bit;
reg [BIT_COUNT_WIDTH_PP-1:0] bit_count;
wire bit_count_done;

//-----
-----
// Functions & Tasks
//-----
-----

function [4*BCD_DIGITS_OUT_PP-1:0] bcd_asl;
    input [4*BCD_DIGITS_OUT_PP-1:0] din;
    input newbit;
    integer k;
    reg cin;
    reg [3:0] digit;
    reg [3:0] digit_less;
    begin
        cin = newbit;

```

```

for (k=0; k<BCD_DIGITS_OUT_PP; k=k+1)
begin
    digit[3] = din[4*k+3];
    digit[2] = din[4*k+2];
    digit[1] = din[4*k+1];
    digit[0] = din[4*k];
    digit_less = digit - 5;
    if (digit > 4'b0100)
    begin
        bcd_asl[4*k+3] = digit_less[2];
        bcd_asl[4*k+2] = digit_less[1];
        bcd_asl[4*k+1] = digit_less[0];
        bcd_asl[4*k+0] = cin;
        cin = 1'b1;
    end
    else
    begin
        bcd_asl[4*k+3] = digit[2];
        bcd_asl[4*k+2] = digit[1];
        bcd_asl[4*k+1] = digit[0];
        bcd_asl[4*k+0] = cin;
        cin = 1'b0;
    end
end // end of for loop
end
endfunction

//-----
//-----
// Module code
//-----
//-----

// Perform proper shifting, binary ASL and BCD ASL
assign bin_next = {bin_reg,1'b0};
always @(bcd_reg or bin_reg)
begin
    bcd_next <= bcd_asl(bcd_reg,bin_reg[BITS_IN_PP-1]);
end

// Busy bit, input and output registers
always @(posedge clk_i)
begin
    if (rst_i)
    begin
        busy_bit <= 0; // Synchronous reset
        dat_bcd_o <= 0;
    end
    else if (start_i && ~busy_bit)
    begin
        busy_bit <= 1;
        bin_reg <= dat_binary_i;
        bcd_reg <= 0;
    end
    else if (busy_bit && ce_i && bit_count_done && ~start_i)
    begin

```

```

    busy_bit <= 0;
    dat_bcd_o <= bcd_next;
end
else if (busy_bit && ce_i && ~bit_count_done)
begin
    bcd_reg <= bcd_next;
    bin_reg <= bin_next;
end
end
assign done_o = ~busy_bit;

// Bit counter
always @(posedge clk_i)
begin
    if (~busy_bit) bit_count <= 0;
    else if (ce_i && ~bit_count_done) bit_count <= bit_count + 1;
end
assign bit_count_done = (bit_count == (BITS_IN_PP-1));

endmodule

```

cdfsm.v

```

module cdfsm(pixel_clock, reset_sync, down_sync,
            i0, i1, i2, i3, j0, j1, j2, j3,
            b0_colored, b1_colored, b2_colored, b3_colored,
            cd_i0, cd_i1, cd_i2, cd_i3,
            collision, final_dist);

    input pixel_clock, reset_sync, down_sync;
    input [4:0] i0, i1, i2, i3;
    input [3:0] j0, j1, j2, j3;

    input      b0_colored, b1_colored, b2_colored, b3_colored;
    output [4:0] cd_i0, cd_i1, cd_i2, cd_i3;

    output collision;
    output [4:0] final_dist;

    reg [2:0] state, next;
    reg collision;
    reg [4:0] temp_dist;
    reg [4:0] final_dist;

    parameter VOID = 3'b00;
    parameter IDLE = 0;
    parameter INCREMENT = 1;
    parameter CHECK = 2;
    parameter WAIT = 3;
    parameter COLLIDE = 4;
    parameter RETAIN = 5;

    wire result0, result1, result2, result3;
    self_overlap so0(i0, i1, i2, i3, j0, j1, j2, j3, cd_i0, j0, result0);
    self_overlap so1(i0, i1, i2, i3, j0, j1, j2, j3, cd_i1, j1, result1);
    self_overlap so2(i0, i1, i2, i3, j0, j1, j2, j3, cd_i2, j2, result2);
    self_overlap so3(i0, i1, i2, i3, j0, j1, j2, j3, cd_i3, j3, result3);

```

```
assign    cd_i0 = i0 + temp_dist;
assign    cd_i1 = i1 + temp_dist;
assign    cd_i2 = i2 + temp_dist;
assign    cd_i3 = i3 + temp_dist;

wire      hit_bottom;
assign    hit_bottom = (cd_i0 >= 24) || (cd_i1 >= 24) || (cd_i2 >= 24)
|| (cd_i3 >= 24);

wire      hit_blocks;
assign    hit_blocks = (b0_colored && !result0) || (b1_colored
&& !result1) ||
                (b2_colored && !result2) || (b3_colored && !result3);

always @ (posedge pixel_clock) begin
    if (reset_sync) state <= IDLE;
    else if (down_sync) state <= INCREMENT;
    else state <= next;
end

always @ (posedge pixel_clock) begin
    if (state == IDLE)
        temp_dist <= 0;
    else if (state == INCREMENT)
        temp_dist <= temp_dist + 1;

    final_dist <= (collision) ? (hit_blocks ? temp_dist-1 : temp_dist)
    : final_dist;
end

always @ (state) begin
    collision = 0;

    case (state)
    IDLE: begin
        next = IDLE;
    end

    INCREMENT: begin
        next = WAIT;
    end

    WAIT: begin
        next = CHECK;
    end

    CHECK: begin
        if (hit_bottom || hit_blocks) next = COLLIDE;

        else next = INCREMENT;
    end

    COLLIDE: begin
        collision = 1;
        next = RETAIN;
    end
end
```

```

    RETAIN: begin
        collision = 1;
        next = IDLE;
    end

    default: next = IDLE;
endcase
end
endmodule

```

cdfsm_tb.v

```

module cdfsm(pixel_clock, reset_sync, down_sync,
            i0, i1, i2, i3, j0, j1, j2, j3,
            b0_colored, b1_colored, b2_colored, b3_colored,
            cd_i0, cd_i1, cd_i2, cd_i3,
            collision, final_dist);

    input pixel_clock, reset_sync, down_sync;
    input [4:0] i0, i1, i2, i3;
    input [3:0] j0, j1, j2, j3;

    input      b0_colored, b1_colored, b2_colored, b3_colored;
    output [4:0] cd_i0, cd_i1, cd_i2, cd_i3;

    output collision;
    output [4:0] final_dist;

    reg [2:0] state, next;
    reg collision;
    reg [4:0] temp_dist;
    reg [4:0] final_dist;

    parameter VOID = 3'b00;
    parameter IDLE = 0;
    parameter INCREMENT = 1;
    parameter CHECK = 2;
    parameter WAIT = 3;
    parameter COLLIDE = 4;
    parameter RETAIN = 5;

    wire result0, result1, result2, result3;
    self_overlap so0(i0, i1, i2, i3, j0, j1, j2, j3, cd_i0, j0, result0);
    self_overlap so1(i0, i1, i2, i3, j0, j1, j2, j3, cd_i1, j1, result1);
    self_overlap so2(i0, i1, i2, i3, j0, j1, j2, j3, cd_i2, j2, result2);
    self_overlap so3(i0, i1, i2, i3, j0, j1, j2, j3, cd_i3, j3, result3);

    assign cd_i0 = i0 + temp_dist;
    assign cd_i1 = i1 + temp_dist;
    assign cd_i2 = i2 + temp_dist;
    assign cd_i3 = i3 + temp_dist;

    wire hit_bottom;
    assign hit_bottom = (cd_i0 >= 24) || (cd_i1 >= 24) || (cd_i2 >= 24)
|| (cd_i3 >= 24);

    wire hit_blocks;

```

```
assign hit_blocks = (b0_colored && !result0) || (b1_colored
&& !result1) ||
    (b2_colored && !result2) || (b3_colored && !result3);

always @ (posedge pixel_clock) begin
    if (reset_sync) state <= IDLE;
    else if (down_sync) state <= INCREMENT;
    else state <= next;
end

always @ (posedge pixel_clock) begin
    if (state == IDLE)
        temp_dist <= 0;
    else if (state == INCREMENT)
        temp_dist <= temp_dist + 1;

    final_dist <= (collision) ? (hit_blocks ? temp_dist-1 : temp_dist)
    : final_dist;
end

always @ (state) begin
    collision = 0;

    case (state)
    IDLE: begin
        next = IDLE;
    end

    INCREMENT: begin
        next = WAIT;
    end

    WAIT: begin
        next = CHECK;
    end

    CHECK: begin
        if (hit_bottom || hit_blocks) next = COLLIDE;

        else next = INCREMENT;
    end

    end

    COLLIDE: begin
        collision = 1;
        next = RETAIN;
    end

    RETAIN: begin
        collision = 1;
        next = IDLE;
    end

    default: next = IDLE;
    endcase
end
```

```
endmodule
```

counter.v

```
module counter(pixel_clock, reset, update_frame, pace,  
              clean_update, coord_update, check_collision,  
drop_row_update, game_update, swap, signal_reset);  
  input pixel_clock, reset, update_frame;  
  input [1:0] pace;  
  output      clean_update, coord_update, check_collision,  
drop_row_update, game_update, swap, signal_reset;  
  
  reg          clean_update;  
  reg          coord_update;  
  reg          check_collision;  
  reg          drop_row_update;  
  reg          game_update;  
  reg          swap;  
  reg          signal_reset;  
  reg [6:0]    count = 0; //max count = 74 < 2^7 = 128  
  
  parameter    SLOW = 74;  
  parameter    MED  = 24;  
  parameter    FAST = 14;  
  
  wire [6:0]   max;  
  assign       max = (pace == 0) ? SLOW : ((pace == 1) ? MED : FAST);  
  
  always @ (posedge pixel_clock) begin  
    if (reset)          count <= 0;  
    else if (update_frame) count <= (count>=max) ? 0 : count+1;  
  
    clean_update <= (update_frame && count==max-5);  
    coord_update <= (update_frame && count==max-4);  
    check_collision <= (update_frame && count==max-3);  
    drop_row_update <= (update_frame && count==max-2);  
    game_update <= (update_frame && count==max-1);  
    swap <= (update_frame && count==max);  
    signal_reset <= (update_frame && count==0);  
  end // always @ (posedge pixel_clock)  
endmodule
```

debounce.v

```
// Switch Debounce Module  
// use your system clock for the clock input  
// to produce a synchronous, debounced output  
module debounce(reset, clock, noisy, clean);  
  parameter DELAY = 270000; // .01 sec with a 27Mhz clock  
  input reset, clock, noisy;  
  output clean;  
  
  reg [18:0] count;  
  reg new, clean;  
  
  always @(posedge clock)  
    if (reset)
```

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

display.v
module display(pixel_clock, pixel, line, disp_type, i, j, bars, rgb);
    input pixel_clock;
    input [9:0] pixel;
    input [9:0] line;

    input [127:0] bars;

    input [2:0] disp_type;
    output [4:0] i;
    output [3:0] j;

    output [23:0] rgb;

    //colors
    parameter RED = 24'hff0000; //I (255,0,0)
    parameter GREEN = 24'h00ff00; //Z (0,255,0)
    parameter BLUE = 24'h0000ff; //S (0,0,255)
    parameter CYAN = 24'h00ffff; //O (0,255,255)
    parameter MAGENTA = 24'hff00ff; //J (255,0,255)
    parameter YELLOW = 24'hffff00; //L (255,255,0)
    parameter GREY = 24'h808080;
    parameter WHITE = 24'hfffffff;
    parameter BLACK = 24'h000000;

    //dimensions
    parameter BORDER_WIDTH = 10'd10;

    //bar
    parameter BAR_START_X = 10'd400;
    parameter BAR_BASE_Y = 10'd300;
    parameter BAR_WIDTH = 10'd10;
    parameter BAR_GAP = 10'd2;

    //parameter
    parameter BLOCK_SIZE = 23;
    parameter BLOCK_SIZE = 10'd16;
    parameter GAME_HEIGHT = 10'd 25*BLOCK_SIZE;
    parameter GAME_WIDTH = 10'd 10*BLOCK_SIZE;

    //shape
```

```
parameter      EMPTY = 0; //BLACK
parameter      I = 1; //RED
parameter      T = 2; //GREY
parameter      O = 3; //CYAN
parameter      L = 4; //YELLOW
parameter      J = 5; //MAGENTA
parameter      S = 6; //BLUE
parameter      Z = 7; //GREEN

wire [23:0]     rgb_b1, rgb_b2, rgb_b3, rgb_b4;
wire [23:0]     rgb_bar1, rgb_bar2, rgb_bar3, rgb_bar4,
               rgb_bar5, rgb_bar6, rgb_bar7, rgb_bar8,
               rgb_bar9, rgb_bar10, rgb_bar11, rgb_bar12,
               rgb_bar13, rgb_bar14, rgb_bar15, rgb_bar16;

//x, y, corner_x, corner_y, width, height, color, rgb
//white border
rect b1(pixel, line, 10'b0, 10'b0, BORDER_WIDTH, 10'd
2*BORDER_WIDTH+GAME_HEIGHT, WHITE, rgb_b1); //left
rect b2(pixel, line, BORDER_WIDTH+GAME_WIDTH, 10'b0, BORDER_WIDTH,
10'd 2*BORDER_WIDTH+GAME_HEIGHT, WHITE, rgb_b2); //right
rect b3(pixel, line, BORDER_WIDTH, 10'b0, GAME_WIDTH, BORDER_WIDTH,
WHITE, rgb_b3); //top
rect b4(pixel, line, BORDER_WIDTH, BORDER_WIDTH+GAME_HEIGHT,
GAME_WIDTH, BORDER_WIDTH, WHITE, rgb_b4); //bottom

// FFT bars
rect bar_thresh(pixel, line, BAR_START_X - 10'd2*BAR_GAP, BAR_BASE_Y
- 8'd100, BAR_GAP, {2'd0, 8'd100}, RED, rgb_bar_threshold);

rect bar1(pixel, line, BAR_START_X + 10'd0*BAR_WIDTH + 10'd0*BAR_GAP,
BAR_BASE_Y - bars[7:0], BAR_WIDTH, {2'd0, bars[7:0]}, GREEN, rgb_bar1);
rect bar2(pixel, line, BAR_START_X + 10'd1*BAR_WIDTH + 10'd1*BAR_GAP,
BAR_BASE_Y - bars[15:8], BAR_WIDTH, {2'd0, bars[15:8]}, GREEN,
rgb_bar2);
rect bar3(pixel, line, BAR_START_X + 10'd2*BAR_WIDTH + 10'd2*BAR_GAP,
BAR_BASE_Y - bars[23:16], BAR_WIDTH, {2'd0, bars[23:16]}, GREEN,
rgb_bar3);
rect bar4(pixel, line, BAR_START_X + 10'd3*BAR_WIDTH + 10'd3*BAR_GAP,
BAR_BASE_Y - bars[31:24], BAR_WIDTH, {2'd0, bars[31:24]}, GREEN,
rgb_bar4);
rect bar5(pixel, line, BAR_START_X + 10'd4*BAR_WIDTH + 10'd4*BAR_GAP,
BAR_BASE_Y - bars[39:32], BAR_WIDTH, {2'd0, bars[39:32]}, GREEN,
rgb_bar5);
rect bar6(pixel, line, BAR_START_X + 10'd5*BAR_WIDTH + 10'd5*BAR_GAP,
BAR_BASE_Y - bars[47:40], BAR_WIDTH, {2'd0, bars[47:40]}, GREEN,
rgb_bar6);
rect bar7(pixel, line, BAR_START_X + 10'd6*BAR_WIDTH + 10'd6*BAR_GAP,
BAR_BASE_Y - bars[55:48], BAR_WIDTH, {2'd0, bars[55:48]}, GREEN,
rgb_bar7);
rect bar8(pixel, line, BAR_START_X + 10'd7*BAR_WIDTH + 10'd7*BAR_GAP,
BAR_BASE_Y - bars[63:56], BAR_WIDTH, {2'd0, bars[63:56]}, GREEN,
rgb_bar8);
rect bar9(pixel, line, BAR_START_X + 10'd8*BAR_WIDTH + 10'd8*BAR_GAP,
BAR_BASE_Y - bars[71:64], BAR_WIDTH, {2'd0, bars[71:64]}, GREEN,
rgb_bar9);
```

```

    rect bar10(pixel, line, BAR_START_X + 10'd9*BAR_WIDTH +
10'd9*BAR_GAP, BAR_BASE_Y - bars[79:72], BAR_WIDTH, {2'd0, bars[79:72]},
GREEN, rgb_bar10);
    rect bar11(pixel, line, BAR_START_X + 10'd10*BAR_WIDTH +
10'd10*BAR_GAP, BAR_BASE_Y - bars[87:80], BAR_WIDTH, {2'd0,
bars[87:80]}, GREEN, rgb_bar11);
    rect bar12(pixel, line, BAR_START_X + 10'd11*BAR_WIDTH +
10'd11*BAR_GAP, BAR_BASE_Y - bars[95:88], BAR_WIDTH, {2'd0,
bars[95:88]}, GREEN, rgb_bar12);
    rect bar13(pixel, line, BAR_START_X + 10'd12*BAR_WIDTH +
10'd12*BAR_GAP, BAR_BASE_Y - bars[103:96], BAR_WIDTH, {2'd0,
bars[103:96]}, GREEN, rgb_bar13);
    rect bar14(pixel, line, BAR_START_X + 10'd13*BAR_WIDTH +
10'd13*BAR_GAP, BAR_BASE_Y - bars[111:104], BAR_WIDTH, {2'd0,
bars[111:104]}, GREEN, rgb_bar14);
    rect bar15(pixel, line, BAR_START_X + 10'd14*BAR_WIDTH +
10'd14*BAR_GAP, BAR_BASE_Y - bars[119:112], BAR_WIDTH, {2'd0,
bars[119:112]}, GREEN, rgb_bar15);
    rect bar16(pixel, line, BAR_START_X + 10'd15*BAR_WIDTH +
10'd15*BAR_GAP, BAR_BASE_Y - bars[127:120], BAR_WIDTH, {2'd0,
bars[127:120]}, GREEN, rgb_bar16);

    reg [23:0]      block_rgb;

    assign      i = (line - BORDER_WIDTH) / BLOCK_SIZE;
    assign      j = (pixel - BORDER_WIDTH) / BLOCK_SIZE;

    always @ (posedge pixel_clock) begin
        if ( ((pixel >= BORDER_WIDTH) && (pixel <
BORDER_WIDTH+GAME_WIDTH)) &&
            ((line >= BORDER_WIDTH) && (line < BORDER_WIDTH+GAME_HEIGHT)) )
        begin

            case (disp_type)
                EMPTY : block_rgb = BLACK;
                I : block_rgb = RED;
                T : block_rgb = GREY;
                O : block_rgb = CYAN;
                L : block_rgb = YELLOW;
                J : block_rgb = MAGENTA;
                S : block_rgb = BLUE;
                Z : block_rgb = GREEN;
                default: block_rgb = BLACK;
            endcase // case(block_type)

        end else begin
            block_rgb = BLACK;
        end
    end

    assign      rgb = rgb_b1 | rgb_b2 | rgb_b3 | rgb_b4 | block_rgb |
                    rgb_bar1 | rgb_bar2 | rgb_bar3 | rgb_bar4 |
                    rgb_bar5 | rgb_bar6 | rgb_bar7 | rgb_bar8 |
                    rgb_bar9 | rgb_bar10 | rgb_bar11 | rgb_bar12 |
                    rgb_bar13 | rgb_bar14 | rgb_bar15 | rgb_bar16;

```

```
endmodule // display_field

display_enable.v
`timescale 1ns / 1ps
////////////////////////////////////
//////////
// Module display_enable - Creates arbitrary timed enable pulses
////////////////////////////////////
//////////
module display_enable(clk, duration, enable, sample_led);
    input clk;
        input [31:0] duration;
    output enable;
        output sample_led;

    reg [31:0] cycle_count = 0;
    reg enable = 0;
    reg sample_led = 0;

    always @(posedge clk)
        begin
            // Disable enable pulse by default
            enable <= 0;
            if (cycle_count == duration) begin
                // Time elapsed; pulse cycle & reset counter
                enable <= 1;
                cycle_count <= 0;
                sample_led = !sample_led;
            end
            else begin
                // Counting: increment cycle_count
                cycle_count <= cycle_count + 1;
            end
        end
    end

endmodule

fancydots.v
`timescale 1ns / 10ps
////////////////////////////////////
//////////
//
// Lab 3 Memory Tester: Number to bitmap decoder
//
// This module converts a 4-bit input to a 80-dot (2 digit) bitmap
// representing
// the numbers ' 0' through '15'.
//
// Author: Yun Wu, Nathan Ickes
// Date: March 8, 2006
//
////////////////////////////////////
//////////

module fancy_dots(clk, num, dots_digits);
    parameter NUM_DIGITS = 4;
```

```

input clk;
input [12:0] num;
output [159:0] dots_digits; //Orig [159:0]

    reg [39:0] dots_1_digit;
    reg [39:0] dots_2_digit;
    reg [39:0] dots_3_digit;
    reg [39:0] dots_4_digit;
    reg [159:0] dots_digits;
    reg [3:0] val1 = 0;
    reg [3:0] val2 = 0;
    reg [3:0] val3 = 0;
    reg [3:0] val4 = 0;
    reg startConversion = 0;

    wire [15:0] bcdVal;
    wire doneConversion;

    binary_to_bcd BinConverter (
        clk,
        1'b1,
        1'b0,
        startConversion,
        num,
        bcdVal,
        doneConversion
    );

    always @ (posedge clk) begin

        if (doneConversion) begin
            if (startConversion) begin
                startConversion <= 0;
            end
            else begin
                val1 = bcdVal[3:0];
                val2 = bcdVal[7:4];
                val3 = bcdVal[11:8];
                val4 = bcdVal[15:12];

                case (val4)
                    4'd09: dots_1_digit <=
{40'b00000110_01001001_01001001_00101001_00011110}; // ' 9'
                    4'd08: dots_1_digit <=
{40'b00110110_01001001_01001001_01001001_00110110}; // ' 8'
                    4'd07: dots_1_digit <=
{40'b00000001_01110001_00001001_00000101_00000011}; // ' 7'
                    4'd06: dots_1_digit <=
{40'b00111100_01001010_01001001_01001001_00110000}; // ' 6'
                    4'd05: dots_1_digit <=
{40'b00100111_01000101_01000101_01000101_00111001}; // ' 5'
                    4'd04: dots_1_digit <=
{40'b00011000_00010100_00010010_01111111_00010000}; // ' 4'
                    4'd03: dots_1_digit <=
{40'b00100010_01000001_01001001_01001001_00110110}; // ' 3'

```

```

        4'd02: dots_1_digit <=
{40'b01100010_01010001_01001001_01001001_01000110}; // ' 2'
        4'd01: dots_1_digit <=
{40'b00000000_01000010_01111111_01000000_00000000}; // ' 1'
        4'd00: dots_1_digit <=
{40'b00111110_01010001_01001001_01000101_00111110}; // ' 0'
        default: dots_1_digit <=
{40'b00000110_01001001_01001001_00101001_00011110}; // ' 9'
    endcase

        case (val3)
            4'd09: dots_2_digit <=
{40'b00000110_01001001_01001001_00101001_00011110}; // ' 9'
            4'd08: dots_2_digit <=
{40'b00110110_01001001_01001001_01001001_00110110}; // ' 8'
            4'd07: dots_2_digit <=
{40'b00000001_01110001_00001001_00000101_00000011}; // ' 7'
            4'd06: dots_2_digit <=
{40'b00111100_01001010_01001001_01001001_00110000}; // ' 6'
            4'd05: dots_2_digit <=
{40'b00100111_01000101_01000101_01000101_00111001}; // ' 5'
            4'd04: dots_2_digit <=
{40'b00011000_00010100_00010010_01111111_00010000}; // ' 4'
            4'd03: dots_2_digit <=
{40'b00100010_01000001_01001001_01001001_00110110}; // ' 3'
            4'd02: dots_2_digit <=
{40'b01100010_01010001_01001001_01001001_01000110}; // ' 2'
            4'd01: dots_2_digit <=
{40'b00000000_01000010_01111111_01000000_00000000}; // ' 1'
            4'd00: dots_2_digit <=
{40'b00111110_01010001_01001001_01000101_00111110}; // ' 0'
            default: dots_2_digit <=
{40'b00000110_01001001_01001001_00101001_00011110}; // ' 9'
        endcase

        case (val2)
            4'd09: dots_3_digit <=
{40'b00000110_01001001_01001001_00101001_00011110}; // ' 9'
            4'd08: dots_3_digit <=
{40'b00110110_01001001_01001001_01001001_00110110}; // ' 8'
            4'd07: dots_3_digit <=
{40'b00000001_01110001_00001001_00000101_00000011}; // ' 7'
            4'd06: dots_3_digit <=
{40'b00111100_01001010_01001001_01001001_00110000}; // ' 6'
            4'd05: dots_3_digit <=
{40'b00100111_01000101_01000101_01000101_00111001}; // ' 5'
            4'd04: dots_3_digit <=
{40'b00011000_00010100_00010010_01111111_00010000}; // ' 4'
            4'd03: dots_3_digit <=
{40'b00100010_01000001_01001001_01001001_00110110}; // ' 3'
            4'd02: dots_3_digit <=
{40'b01100010_01010001_01001001_01001001_01000110}; // ' 2'
            4'd01: dots_3_digit <=
{40'b00000000_01000010_01111111_01000000_00000000}; // ' 1'
            4'd00: dots_3_digit <=
{40'b00111110_01010001_01001001_01000101_00111110}; // ' 0'

```

```

                default: dots_3_digit <=
{40'b00000110_01001001_01001001_00101001_00011110}; // ' 9'
                endcase

                case (vall)
                    4'd09: dots_4_digit <=
{40'b00000110_01001001_01001001_00101001_00011110}; // ' 9'
                    4'd08: dots_4_digit <=
{40'b00110110_01001001_01001001_01001001_00110110}; // ' 8'
                    4'd07: dots_4_digit <=
{40'b00000001_01110001_00001001_00000101_00000011}; // ' 7'
                    4'd06: dots_4_digit <=
{40'b00111100_01001010_01001001_01001001_00110000}; // ' 6'
                    4'd05: dots_4_digit <=
{40'b00100111_01000101_01000101_01000101_00111001}; // ' 5'
                    4'd04: dots_4_digit <=
{40'b00011000_00010100_00010010_01111111_00010000}; // ' 4'
                    4'd03: dots_4_digit <=
{40'b00100010_01000001_01001001_01001001_00110110}; // ' 3'
                    4'd02: dots_4_digit <=
{40'b01100010_01010001_01001001_01001001_01000110}; // ' 2'
                    4'd01: dots_4_digit <=
{40'b00000000_01000010_01111111_01000000_00000000}; // ' 1'
                    4'd00: dots_4_digit <=
{40'b00111110_01010001_01001001_01000101_00111110}; // ' 0'
                default: dots_4_digit <=
{40'b00000110_01001001_01001001_00101001_00011110}; // ' 9'
                endcase

                dots_digits = {dots_1_digit, dots_2_digit,
dots_3_digit, dots_4_digit};

                startConversion <= 1;

            end
        end

    end

endmodule

```

labkit.v

```

`timescale 1ns / 10ps

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

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

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

```

```
tv_in_ycrCb, tv_in_data_valid, tv_in_line_clock1,  
tv_in_line_clock2, tv_in_aef, tv_in_hff, tv_in_aff,  
tv_in_i2c_clock, tv_in_i2c_data, tv_in_fifo_read,  
tv_in_fifo_clock, tv_in_iso, tv_in_reset_b, tv_in_clock,  
  
ram0_data, ram0_address, ram0_adv_ld, ram0_clk, ram0_cen_b,  
ram0_ce_b, ram0_oe_b, ram0_we_b, ram0_bwe_b,  
  
ram1_data, ram1_address, ram1_adv_ld, ram1_clk, ram1_cen_b,  
ram1_ce_b, ram1_oe_b, ram1_we_b, ram1_bwe_b,  
  
clock_feedback_out, clock_feedback_in,  
  
flash_data, flash_address, flash_ce_b, flash_oe_b,  
flash_we_b,  
flash_reset_b, flash_sts, flash_byte_b,  
  
rs232_txd, rs232_rxd, rs232_rts, rs232_cts,  
  
mouse_clock, mouse_data, keyboard_clock, keyboard_data,  
  
clock_27mhz, clock1, clock2,  
  
disp_blank, disp_data_out, disp_clock, disp_rs, disp_ce_b,  
disp_reset_b, disp_data_in,  
  
button0, button1, button2, button3, button_enter,  
button_right,  
button_left, button_down, button_up,  
  
switch,  
  
led,  
  
user1, user2, user3, user4,  
  
daughtercard,  
  
systemace_data, systemace_address, systemace_ce_b,  
systemace_we_b, systemace_oe_b, systemace_irq,  
systemace_mpbrdy,  
  
analyzer1_data, analyzer1_clock,  
analyzer2_data, analyzer2_clock,  
analyzer3_data, analyzer3_clock,  
analyzer4_data, analyzer4_clock);  
  
output beep, audio_reset_b, ac97_synch, ac97_sdata_out;  
input ac97_bit_clock, ac97_sdata_in;  
  
output [7:0] vga_out_red, vga_out_green, vga_out_blue;  
output vga_out_sync_b, vga_out_blank_b, vga_out_pixel_clock,  
vga_out_hsync, vga_out_vsync;  
  
output [9:0] tv_out_ycrCb;
```

```
output tv_out_reset_b, tv_out_clock, tv_out_i2c_clock,
tv_out_i2c_data,
    tv_out_pal_ntsc, tv_out_hsync_b, tv_out_vsync_b, tv_out_blank_b,
    tv_out_subcar_reset;

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

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

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

input  clock_feedback_in;
output clock_feedback_out;

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

output rs232_txd, rs232_rts;
input  rs232_rxd, rs232_cts;

input  mouse_clock, mouse_data, keyboard_clock, keyboard_data;

input  clock_27mhz, clock1, clock2;

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

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

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

inout [43:0] daughtercard;

inout  [15:0] systemace_data;
output [6:0]  systemace_address;
```

```
output systemace_ce_b, systemace_we_b, systemace_oe_b;
input  systemace_irq, systemace_mpbrdy;

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

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

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

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

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

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

// SRAMs
assign ram0_data = 36'hZ;
assign ram0_address = 19'h0;
assign ram0_adv_ld = 1'b0;
assign ram0_clk = 1'b0;
assign ram0_cen_b = 1'b1;
assign ram0_ce_b = 1'b1;
assign ram0_oe_b = 1'b1;
assign ram0_we_b = 1'b1;
assign ram0_bwe_b = 4'hF;
assign ram1_data = 36'hZ;
assign ram1_address = 19'h0;
```

```
assign ram1_adv_ld = 1'b0;
assign ram1_clk = 1'b0;
assign ram1_cen_b = 1'b1;
assign ram1_ce_b = 1'b1;
assign ram1_oe_b = 1'b1;
assign ram1_we_b = 1'b1;
assign ram1_bwe_b = 4'hF;
assign clock_feedback_out = 1'b0;

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

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

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

// User I/Os
assign user1[31:3] = 30'hZ;
assign user1[1:0] = 2'hZ;
assign user2 = 32'hZ;
assign user3 = 32'hZ;
assign user4 = 32'hZ;

// Daughtercard Connectors
assign daughtercard = 44'hZ;

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

////////////////////////////////////
////
//
// Laser Tetris Components
//

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

//
// Generate a 31.5MHz pixel clock from clock_27mhz
//
```

```

wire  pclk, pixel_clock;
wire  reset;
wire  [9:0] pixel_count;
wire  [9:0] line_count;
wire  laser_hsync;
wire  laser_vsync;
wire  laser_output;
wire  [7:0] from_ac97_data, to_ac97_data;
wire  ready;
wire  playback;

wire  event_enable;

DCM pixel_clock_dcm (.CLKIN(clock_27mhz), .CLKFX(pclk));
// synthesis attribute CLKFX_DIVIDE of pixel_clock_dcm is 6
// synthesis attribute CLKFX_MULTIPLY of pixel_clock_dcm is 7
// synthesis attribute CLK_FEEDBACK of pixel_clock_dcm is "NONE"
BUFG pixel_clock_buf (.I(pclk), .O(pixel_clock));

assign analyzer4_data = 16'h0;
assign analyzer4_clock = 1'b1;

assign reset = ~button0;

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

assign laser_hsync = user1[0];
assign laser_vsync = user1[1];
assign user1[2] = laser_output | switch[4];
assign led[0] = laser_hsync; // laser hsync
assign led[1] = laser_vsync; // laser vsync
assign led[2] = ~event_enable;
assign led[6:3] = 5'b1111;

////////////////////////////////////
////////////////////////////////////
/* James' Stuff */

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

//debounce modules
wire reset_sync, up_sync, down_sync, left_sync, right_sync;
debounce
db_reset(.reset(1'b0), .clock(pixel_clock), .noisy(~button_enter), .cle
an(reset_sync));
debounce
db_up(.reset(reset_sync), .clock(pixel_clock), .noisy(~button_up), .cle
an(up_sync));
debounce
db_down(.reset(reset_sync), .clock(pixel_clock), .noisy(~button_down),
.clean(down_sync));

```

```

    debounce
db_left(.reset(reset_sync), .clock(pixel_clock), .noisy(~button_left),
.clean(left_sync));
    debounce
db_right(.reset(reset_sync), .clock(pixel_clock), .noisy(~button_right)
, .clean(right_sync));

    wire [23:0] rgb;
    wire      update_frame;
    vga my_vga(pixel_clock, reset_sync, vga_out_hsync, vga_out_vsync,
vga_out_sync_b,
            vga_out_blank_b, pixel_count, line_count, update_frame);

    wire      clean_update, coord_update, check_collision,
drop_row_update, game_update, swap, signal_reset;
    wire [1:0] pace;
    assign    pace = switch[3] ? (event_enable ? 2'b10 : 2'b00) :
switch[1:0];
    counter mycounter(pixel_clock, reset_sync, update_frame, pace,

            clean_update, coord_update, check_collision,
drop_row_update, game_update, swap, signal_reset);

    wire      rotate, left, right;
    signal_reg rotate_reg(reset_sync, signal_reset, up_sync, rotate);
    signal_reg left_reg(reset_sync, signal_reset, left_sync, left);
    signal_reg right_reg(reset_sync, signal_reset, right_sync, right);

    wire [2:0] disp_type;
    wire [4:0] i;
    wire [3:0] j;

    wire [4:0] i0, i1, i2, i3;
    wire [3:0] j0, j1, j2, j3;
    wire [2:0] drop_type;

    wire [7:0] pd_out;
    wire      ce;
    wire      new_block;
    wire [2:0] rand_type;
    wire      new_type;

    assign    change = reset_sync || (game_update && new_block);
    rand myrand(pixel_clock, pd_out, change);
    assign    rand_type = (pd_out[2:0] == 3'b000) ? 3'b001 :
pd_out[2:0];
    assign    drop_type = switch[2] ? rand_type : switch[7:5];

    wire      b0_colored, b1_colored, b2_colored, b3_colored;
    wire      next0_colored, next1_colored, next2_colored,
next3_colored;
    wire [4:0] cd_i0, cd_i1, cd_i2, cd_i3;
    wire      collision, drop;
    wire [4:0] final_dist;

    wire [127:0] bars;

```

```
cd fsm collision_detector(pixel_clock, reset_sync, down_sync,
                          i0, i1, i2, i3, j0, j1, j2, j3,
                          b0_colored, b1_colored, b2_colored, b3_colored,
                          cd_i0, cd_i1, cd_i2, cd_i3,
                          collision, final_dist);

signal_reg collision_reg(reset_sync, signal_reset, collision, drop);

display disp(pixel_clock, pixel_count, line_count, disp_type, i, j,
bars, rgb);

wire [12:0] score;
wire [4:0] temp_i0, temp_i1, temp_i2, temp_i3;
wire [3:0] temp_j0, temp_j1, temp_j2, temp_j3;
wire cc0_colored, cc1_colored, cc2_colored, cc3_colored;

minor fsm drop_controller(pixel_clock, reset_sync, rotate, drop, left,
right,
                          coord_update, check_collision, drop_row_update,
swap, signal_reset,
                          drop_type, final_dist,
next0_colored, next1_colored, next2_colored,
next3_colored,
                          cc0_colored, cc1_colored, cc2_colored,
cc3_colored,
                          new_block, i0, i1, i2, i3, j0, j1, j2, j3,
temp_i0, temp_i1, temp_i2, temp_i3,
temp_j0, temp_j1, temp_j2, temp_j3,
score);

wire [4:0] laser_i;
wire [3:0] laser_j;
wire laser_disp_type;

major fsm map_controller(pixel_clock, reset_sync, clean_update,
game_update,
                          i0, i1, i2, i3, j0, j1, j2, j3, drop_type,
new_block,
                          i, j, laser_i, laser_j, cd_i0, cd_i1, cd_i2, cd_i3,
temp_i0, temp_i1, temp_i2, temp_i3,
temp_j0, temp_j1, temp_j2, temp_j3,
                          disp_type, laser_disp_type, b0_colored, b1_colored,
b2_colored, b3_colored,
                          next0_colored, next1_colored, next2_colored,
next3_colored,
                          cc0_colored, cc1_colored, cc2_colored,
cc3_colored);

assign vga_out_red = rgb[23:16];
assign vga_out_green = rgb[15:8];
assign vga_out_blue = rgb[7:0];
```

```
////////////////////////////////////
////////////////////////////////////
/* Cameron's Stuff */
////////////////////////////////////
////////////////////////////////////

wire [25:0]    fft_mag;

wire [9:0]     xk_index;
wire [18:0]    xk_re;
wire [18:0]    xk_im;
wire          audio_clk;

assign analyzer1_data = {xk_index, 14'd0};
assign analyzer1_clock = pixel_clock;
assign analyzer2_data = {xk_re, 3'd0};
assign analyzer2_clock = pixel_clock;
assign analyzer3_data = {xk_im, 3'd0};
assign analyzer3_clock = pixel_clock;

audio_processor A_PROC (
    ac97_bit_clock,
    from_ac97_data,
    ready,
    bars,
    xk_index,
    xk_re,
    xk_im,
    event_enable
);

// AC97 driver
audio a(clock_27mhz, reset, from_ac97_data, to_ac97_data, ready,
    audio_reset_b, ac97_sdata_out, ac97_sdata_in,
    ac97_synch, ac97_bit_clock);

// record module
recorder r(clock_27mhz, reset, playback, ready, from_ac97_data,
to_ac97_data);

// Instantiate the Top Module from the schematic
top TopModule (
    .reset(reset),
    .up(~button_up),
    .down(~button_down),
    .game_speed(switch[3:0]),
    .laser_hsync(laser_hsync),
    .laser_vsync(laser_vsync),
    .laser_output(laser_output),
    .disp_test(1'b0),
    .disp_blank(disp_blank),
    .disp_clock(disp_clock),
    .disp_rs(disp_rs),
    .disp_ce_b(disp_ce_b),
    .disp_reset_b(disp_reset_b),
    .disp_data_out(disp_data_out),
    .sample_led(led[7]),
```

```
        .laser_i(laser_i),
        .laser_j(laser_j),
        .laser_disp_type(laser_disp_type),
        .score(score),
        .pixel_clock(pixel_clock)
    );
endmodule

majorfsm.v
module majorfsm(pixel_clock, reset, clean_update, game_update,
    i0, i1, i2, i3, j0, j1, j2, j3, drop_type, new_block,
    i, j, laser_i, laser_j, cd_i0, cd_i1, cd_i2, cd_i3,
    temp_i0, temp_i1, temp_i2, temp_i3,
    temp_j0, temp_j1, temp_j2, temp_j3,
    disp_type, laser_disp_type, b0_colored, b1_colored,
b2_colored, b3_colored,
    next0_colored, next1_colored, next2_colored, next3_colored,
    cc0_colored, cc1_colored, cc2_colored, cc3_colored);

    input pixel_clock;
    input reset;
    input clean_update, game_update;

    input [4:0] i0, i1, i2, i3;
    input [3:0] j0, j1, j2, j3;
    input [2:0] drop_type;
    input      new_block;

    input [4:0] i, cd_i0, cd_i1, cd_i2, cd_i3;
    input [3:0] j;

    input [4:0] laser_i;
    input [3:0] laser_j;

    input [4:0] temp_i0, temp_i1, temp_i2, temp_i3;
    input [3:0] temp_j0, temp_j1, temp_j2, temp_j3;

    output [2:0] disp_type;
    output  laser_disp_type;
    output  b0_colored, b1_colored, b2_colored, b3_colored;
    output  next0_colored, next1_colored, next2_colored, next3_colored;
    output  cc0_colored, cc1_colored, cc2_colored, cc3_colored;

    reg [2:0]      map [24:0][9:0];

    assign  disp_type = map[i][j];

    assign  laser_disp_type = (map[laser_i][laser_j] == 3'b000) ? 0 : 1;

    assign  b0_colored = (map[cd_i0][j0] != 3'b000);
    assign  b1_colored = (map[cd_i1][j1] != 3'b000);
    assign  b2_colored = (map[cd_i2][j2] != 3'b000);
    assign  b3_colored = (map[cd_i3][j3] != 3'b000);

    assign  next0_colored = (map[i0+1][j0] != 3'b000);
    assign  next1_colored = (map[i1+1][j1] != 3'b000);
    assign  next2_colored = (map[i2+1][j2] != 3'b000);
```

```
assign    next3_colored = (map[i3+1][j3] != 3'b000);

assign    cc0_colored = (map[temp_i0][temp_j0] != 3'b000);
assign    cc1_colored = (map[temp_i1][temp_j1] != 3'b000);
assign    cc2_colored = (map[temp_i2][temp_j2] != 3'b000);
assign    cc3_colored = (map[temp_i3][temp_j3] != 3'b000);

integer  m, n;

always @ (posedge pixel_clock) begin
    if (reset) begin
        for (m=0; m<25; m=m+1) begin
            for (n=0; n<10; n=n+1) begin
                map[m][n] <= 0;
            end
        end
    end else if (clean_update && ! new_block) begin

        if ( ((i0 >= 0) && (i0 <= 24)) &&
              ((j0 >= 0) && (j0 <= 9)) ) begin
            map[i0][j0] <= 0;
        end

        if ( ((i1 >= 0) && (i1 <= 24)) &&
              ((j1 >= 0) && (j1 <= 9)) ) begin
            map[i1][j1] <= 0;
        end

        if ( ((i2 >= 0) && (i2 <= 24)) &&
              ((j2 >= 0) && (j2 <= 9)) ) begin
            map[i2][j2] <= 0;
        end

        if ( ((i3 >= 0) && (i3 <= 24)) &&
              ((j3 >= 0) && (j3 <= 9)) ) begin
            map[i3][j3] <= 0;
        end

    end else if (game_update) begin
        if ( ((i0 >= 0) && (i0 <= 24)) &&
              ((j0 >= 0) && (j0 <= 9)) ) begin
            map[i0][j0] <= drop_type;
        end

        if ( ((i1 >= 0) && (i1 <= 24)) &&
              ((j1 >= 0) && (j1 <= 9)) ) begin
            map[i1][j1] <= drop_type;
        end

        if ( ((i2 >= 0) && (i2 <= 24)) &&
              ((j2 >= 0) && (j2 <= 9)) ) begin
            map[i2][j2] <= drop_type;
        end

        if ( ((i3 >= 0) && (i3 <= 24)) &&
              ((j3 >= 0) && (j3 <= 9)) ) begin
            map[i3][j3] <= drop_type;
        end
    end
end
```

```
        end
    end
end
endmodule
```

minorfsm.v

```
module minorfsm(pixel_clock, reset, rotate, drop, left, right,
                coord_update, check_collision, drop_row_update, swap,
signal_reset,
                drop_type, final_dist,
                next0_colored, next1_colored, next2_colored, next3_colored,
                cc0_colored, cc1_colored, cc2_colored, cc3_colored,
                new_block, i0, i1, i2, i3, j0, j1, j2, j3,
                temp_i0, temp_i1, temp_i2, temp_i3,
                temp_j0, temp_j1, temp_j2, temp_j3,
                score);

    input pixel_clock;
    input reset, rotate, drop, left, right;
    input coord_update, check_collision, drop_row_update, swap,
signal_reset;
    input [2:0] drop_type;
    input [4:0] final_dist;
    input      next0_colored, next1_colored, next2_colored,
next3_colored;
    input      cc0_colored, cc1_colored, cc2_colored, cc3_colored;

    output [4:0] i0, i1, i2, i3;
    output [3:0] j0, j1, j2, j3;
    output [4:0] temp_i0, temp_i1, temp_i2, temp_i3;
    output [3:0] temp_j0, temp_j1, temp_j2, temp_j3;

    output  new_block;
    reg      new_block;

    output [12:0] score;
    reg [12:0]      score, temp_score;

    reg [4:0]      i0, i1, i2, i3;
    reg [3:0]      j0, j1, j2, j3;

    reg [4:0]      temp_i0, temp_i1, temp_i2, temp_i3;
    reg [3:0]      temp_j0, temp_j1, temp_j2, temp_j3;

    reg            vertical = 1;
    reg            temp_vertical;
    reg [2:0]      move;
    reg            gameover;

    //parameter    BLACK = 24'h000000;
    parameter      MAX_J = 9;
    parameter      SPEED = 1;
    parameter      STAY = 0;
    parameter      GOLEFT = 1;
    parameter      GORIGHT = 2;
    parameter      DOROTATE = 3;
    parameter      DODROP = 4;
```

```

//block_type
parameter      I = 1; //RED
parameter      T = 2; //GREY
parameter      O = 3; //CYAN
parameter      L = 4; //YELLOW
parameter      J = 5; //MAGENTA
parameter      S = 6; //BLUE
parameter      Z = 7; //GREEN

wire [4:0]      i0_1_ortho, i2_1_ortho, i3_1_ortho, i3_1_dia,
i0_3_ortho, i2_3_ortho, i3_3_dia, i1_2_ortho, i3_2_ortho, i0_2_dia;
wire [3:0]      j0_1_ortho, j2_1_ortho, j3_1_ortho, j3_1_dia,
j0_3_ortho, j2_3_ortho, j3_3_dia, j1_2_ortho, j3_2_ortho, j0_2_dia;

//I and O are special rotational cases
//following dictate rotational behaviors of T, L, J (center 1)
rotate_ortho ortho_0_1(i0, j0, i1, j1, i0_1_ortho, j0_1_ortho); //T,
L, J
rotate_ortho ortho_2_1(i2, j2, i1, j1, i2_1_ortho, j2_1_ortho); //T,
L, J
rotate_ortho ortho_3_1(i3, j3, i1, j1, i3_1_ortho, j3_1_ortho); //T
rotate_dia dia_3_1(i3, j3, i1, j1, i3_1_dia, j3_1_dia); //L, J

//S (center 3)
rotate_ortho ortho_0_3(i0, j0, i3, j3, i0_3_ortho, j0_3_ortho);
rotate_ortho ortho_2_3(i2, j2, i3, j3, i2_3_ortho, j2_3_ortho);
rotate_dia dia_1_3(i1, j1, i3, j3, i3_3_dia, j3_3_dia);

//Z (center 2)
rotate_ortho ortho_1_2(i1, j1, i2, j2, i1_2_ortho, j1_2_ortho);
rotate_ortho ortho_3_2(i3, j3, i2, j2, i3_2_ortho, j3_2_ortho);
rotate_dia dia_0_2(i0, j0, i2, j2, i0_2_dia, j0_2_dia);

wire      result0, result1, result2, result3;
self_overlap so0(i0, i1, i2, i3, j0, j1, j2, j3, i0+5'd1, j0,
result0);
self_overlap so1(i0, i1, i2, i3, j0, j1, j2, j3, i1+5'd1, j1,
result1);
self_overlap so2(i0, i1, i2, i3, j0, j1, j2, j3, i2+5'd1, j2,
result2);
self_overlap so3(i0, i1, i2, i3, j0, j1, j2, j3, i3+5'd1, j3,
result3);

wire      cc0_so, cc1_so, cc2_so, cc3_so;
self_overlap so4(i0, i1, i2, i3, j0, j1, j2, j3, temp_i0, temp_j0,
cc0_so);
self_overlap so5(i0, i1, i2, i3, j0, j1, j2, j3, temp_i1, temp_j1,
cc1_so);
self_overlap so6(i0, i1, i2, i3, j0, j1, j2, j3, temp_i2, temp_j2,
cc2_so);
self_overlap so7(i0, i1, i2, i3, j0, j1, j2, j3, temp_i3, temp_j3,
cc3_so);

wire      hit_bottom;

```

```

assign    hit_bottom = (i0 >= 24) || (i1 >= 24) || (i2 >= 24) || (i3
>= 24);
wire      hit_blocks;
assign    hit_blocks = (next0_colored && !result0) || (next1_colored
&& !result1) ||
          (next2_colored && !result2) || (next3_colored && !result3);

always @ (posedge pixel_clock) begin
    if (left)
        move <= GOLEFT;
    else if (right)
        move <= GORIGHT;
    else if (rotate)
        move <= DOROTATE;
    else if (drop)
        move <= DODROP;
    else move <= STAY;
end

always @ (posedge pixel_clock) begin
    if (reset) begin
        new_block <= 1;
        score <= 0;
        temp_score <= 0;
        gameover <= 0;
        case (drop_type)
            I: {temp_i0, temp_j0, temp_i1, temp_j1, temp_i2, temp_j2,
temp_i3, temp_j3} <= {5'd0, 4'd4, 5'd1, 4'd4, 5'd2, 4'd4, 5'd3, 4'd4};

            T: {temp_i0, temp_j0, temp_i1, temp_j1, temp_i2, temp_j2,
temp_i3, temp_j3} <= {5'd0, 4'd3, 5'd0, 4'd4, 5'd0, 4'd5, 5'd1, 4'd4};
            O: {temp_i0, temp_j0, temp_i1, temp_j1, temp_i2, temp_j2,
temp_i3, temp_j3} <= {5'd0, 4'd4, 5'd0, 4'd5, 5'd1, 4'd4, 5'd1, 4'd5};

            L: {temp_i0, temp_j0, temp_i1, temp_j1, temp_i2, temp_j2,
temp_i3, temp_j3} <= {5'd0, 4'd4, 5'd1, 4'd4, 5'd2, 4'd4, 5'd2, 4'd5};
            J: {temp_i0, temp_j0, temp_i1, temp_j1, temp_i2, temp_j2,
temp_i3, temp_j3} <= {5'd0, 4'd5, 5'd1, 4'd5, 5'd2, 4'd5, 5'd2, 4'd4};
            S: {temp_i0, temp_j0, temp_i1, temp_j1, temp_i2, temp_j2,
temp_i3, temp_j3} <= {5'd0, 4'd4, 5'd0, 4'd5, 5'd1, 4'd3, 5'd1, 4'd4};
            Z: {temp_i0, temp_j0, temp_i1, temp_j1, temp_i2, temp_j2,
temp_i3, temp_j3} <= {5'd0, 4'd3, 5'd0, 4'd4, 5'd1, 4'd4, 5'd1, 4'd5};
            //invalid values of i and j
            default: {i0, j0, i1, j1, i2, j2, i3, j3} <= {5'd25, 4'd10,
5'd25, 4'd10, 5'd25, 4'd10, 5'd25, 4'd10};
        endcase // case(block_type)
    end else if (gameover) begin
        i0 <= i0;
        i1 <= i1;
        i2 <= i2;
        i3 <= i3;
        j0 <= j0;
        j1 <= j1;
        j2 <= j2;
        j3 <= j3;
    end else if (signal_reset && new_block) begin
        temp_score <= temp_score+10;
    end
end

```

```

        case (drop_type)
            I: {temp_i0, temp_j0, temp_i1, temp_j1, temp_i2, temp_j2,
temp_i3, temp_j3} <= {5'd0, 4'd4, 5'd1, 4'd4, 5'd2, 4'd4, 5'd3, 4'd4};

            T: {temp_i0, temp_j0, temp_i1, temp_j1, temp_i2, temp_j2,
temp_i3, temp_j3} <= {5'd0, 4'd3, 5'd0, 4'd4, 5'd0, 4'd5, 5'd1, 4'd4};
            O: {temp_i0, temp_j0, temp_i1, temp_j1, temp_i2, temp_j2,
temp_i3, temp_j3} <= {5'd0, 4'd4, 5'd0, 4'd5, 5'd1, 4'd4, 5'd1, 4'd5};

            L: {temp_i0, temp_j0, temp_i1, temp_j1, temp_i2, temp_j2,
temp_i3, temp_j3} <= {5'd0, 4'd4, 5'd1, 4'd4, 5'd2, 4'd4, 5'd2, 4'd5};
            J: {temp_i0, temp_j0, temp_i1, temp_j1, temp_i2, temp_j2,
temp_i3, temp_j3} <= {5'd0, 4'd5, 5'd1, 4'd5, 5'd2, 4'd5, 5'd2, 4'd4};
            S: {temp_i0, temp_j0, temp_i1, temp_j1, temp_i2, temp_j2,
temp_i3, temp_j3} <= {5'd0, 4'd4, 5'd0, 4'd5, 5'd1, 4'd3, 5'd1, 4'd4};
            Z: {temp_i0, temp_j0, temp_i1, temp_j1, temp_i2, temp_j2,
temp_i3, temp_j3} <= {5'd0, 4'd3, 5'd0, 4'd4, 5'd1, 4'd4, 5'd1, 4'd5};
            //invalid values of i and j
            default:{temp_i0, temp_j0, temp_i1, temp_j1, temp_i2, temp_j2,
temp_i3, temp_j3} <= {5'd25, 4'd10, 5'd25, 4'd10, 5'd25, 4'd10, 5'd25,
4'd10};
        endcase // case(block_type)

    end else if (coord_update && (!new_block) && (!gameover)) begin

        if (move == DODROP) begin
            temp_score <= temp_score + final_dist;
            temp_i0 <= i0 + final_dist;
            temp_i1 <= i1 + final_dist;
            temp_i2 <= i2 + final_dist;
            temp_i3 <= i3 + final_dist;
        end else if ((move == GOLEFT) && (temp_j0 * temp_j1 * temp_j2 *
temp_j3 != 0)) begin
            temp_j0 <= j0 - SPEED;
            temp_j1 <= j1 - SPEED;
            temp_j2 <= j2 - SPEED;
            temp_j3 <= j3 - SPEED;
        end else if ((move == GORIGHT) && (temp_j0 < MAX_J) && (temp_j1
< MAX_J) && (temp_j2 < MAX_J) && (temp_j3 < MAX_J)) begin

            temp_j0 <= j0 + SPEED;
            temp_j1 <= j1 + SPEED;
            temp_j2 <= j2 + SPEED;
            temp_j3 <= j3 + SPEED;
        end else if (move == DOROTATE && drop_type != 0) begin
            case (drop_type)
                I: begin
                    if (vertical) begin
                        temp_i0 <= i1;
                        temp_j0 <= j1-1;
                        temp_i1 <= i1;
                        temp_j1 <= j1;
                        temp_i2 <= i1;
                        temp_j2 <= j1+1;
                        temp_i3 <= i1;
                        temp_j3 <= j1+2;
                        temp_vertical <= 0;
                    end
                end
            endcase
        end
    end
end

```

```
end else begin

    temp_i0 <= i1-1;
    temp_j0 <= j1;
    temp_i1 <= i1;
    temp_j1 <= j1;
    temp_i2 <= i1+1;
    temp_j2 <= j1;
    temp_i3 <= i1+2;
    temp_j3 <= j1;
    temp_vertical <= 1;
end
end

T: begin
    temp_i0 <= i0_1_ortho;
    temp_j0 <= j0_1_ortho;
    temp_i1 <= i1;
    temp_j1 <= j1;
    temp_i2 <= i2_1_ortho;
    temp_j2 <= j2_1_ortho;
    temp_i3 <= i3_1_ortho;
    temp_j3 <= j3_1_ortho;
end

L: begin
    temp_i0 <= i0_1_ortho;
    temp_j0 <= j0_1_ortho;
    temp_i1 <= i1;
    temp_j1 <= j1;
    temp_i2 <= i2_1_ortho;
    temp_j2 <= j2_1_ortho;
    temp_i3 <= i3_1_dia;
    temp_j3 <= j3_1_dia;
end

J: begin
    temp_i0 <= i0_1_ortho;
    temp_j0 <= j0_1_ortho;
    temp_i1 <= i1;
    temp_j1 <= j1;
    temp_i2 <= i2_1_ortho;
    temp_j2 <= j2_1_ortho;
    temp_i3 <= i3_1_dia;
    temp_j3 <= j3_1_dia;
end

S: begin
    temp_i0 <= i0_3_ortho;
    temp_j0 <= j0_3_ortho;
    temp_i1 <= i3_3_dia;
    temp_j1 <= j3_3_dia;
    temp_i2 <= i2_3_ortho;
    temp_j2 <= j2_3_ortho;
    temp_i3 <= i3;
    temp_j3 <= j3;
end
```

```
Z: begin
  temp_i0 <= i0_2_dia;
  temp_j0 <= j0_2_dia;
  temp_i1 <= i1_2_ortho;
  temp_j1 <= j1_2_ortho;
  temp_i2 <= i2;
  temp_j2 <= j2;
  temp_i3 <= i3_2_ortho;
  temp_j3 <= j3_2_ortho;
end

default: begin //invalid values of i and j

  i0 <= i0;
  j0 <= j0;
  i1 <= i1;
  j1 <= j1;
  i2 <= i2;
  j2 <= j2;
  i3 <= i3;
  j3 <= j3;
end
endcase // case(block_type)
end
end else if (check_collision && !gameover) begin
  if (!( (cc0_colored && !cc0_so) ||
        (cc1_colored && !cc1_so) ||
        (cc2_colored && !cc2_so) ||
        (cc3_colored && !cc3_so) ) ) begin

    if ( (temp_i0 >= 0 && temp_i0 <= 24) &&
        (temp_i1 >= 0 && temp_i1 <= 24) &&
        (temp_i2 >= 0 && temp_i2 <= 24) &&
        (temp_i3 >= 0 && temp_i3 <= 24) ) begin

      if ( (temp_j0 >= 0 && temp_j0 <= MAX_J) &&
          (temp_j1 >= 0 && temp_j1 <= MAX_J) &&
          (temp_j2 >= 0 && temp_j2 <= MAX_J) &&
          (temp_j3 >= 0 && temp_j3 <= MAX_J) ) begin

        score <= temp_score;
        vertical <= temp_vertical;
        i0 <= temp_i0;
        i1 <= temp_i1;
        i2 <= temp_i2;
        i3 <= temp_i3;
        j0 <= temp_j0;
        j1 <= temp_j1;
        j2 <= temp_j2;
        j3 <= temp_j3;
      end
    end
  end else if (new_block) begin
    gameover <= 1;
    new_block <= 0;
  end
end
```

```
        i0 <= temp_i0;
        i1 <= temp_i1;
        i2 <= temp_i2;
        i3 <= temp_i3;
        j0 <= temp_j0;
        j1 <= temp_j1;
        j2 <= temp_j2;
        j3 <= temp_j3;
        score <= temp_score;
    end
end else if (drop_row_update && !gameover) begin
    if (new_block == 1) new_block <= 0;
    else begin
        if (hit_bottom || hit_blocks || drop) begin
            score <= score + {8'b0, i0};
            new_block <= 1;
        end else begin
            i0 <= i0 + SPEED;
            i1 <= i1 + SPEED;
            i2 <= i2 + SPEED;
            i3 <= i3 + SPEED;
        end
    end
end
end else if (swap && !gameover) begin
    temp_i0 <= i0;
    temp_i1 <= i1;
    temp_i2 <= i2;
    temp_i3 <= i3;
    temp_j0 <= j0;
    temp_j1 <= j1;
    temp_j2 <= j2;
    temp_j3 <= j3;
    temp_score <= score;
end
end
endmodule
```

rand.v

```
/*
*****
* This file is owned and controlled by Xilinx and must be used
*
* solely for design, simulation, implementation and creation of
*
* design files limited to Xilinx devices or technologies. Use
*
* with non-Xilinx devices or technologies is expressly prohibited
*
* and immediately terminates your license.
*
*
*
* XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
*
* SOLELY FOR USE IN DEVELOPING PROGRAMS AND SOLUTIONS FOR
*
*
*/
```

```
* XILINX DEVICES. BY PROVIDING THIS DESIGN, CODE, OR INFORMATION
*
* AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE, APPLICATION
*
* OR STANDARD, XILINX IS MAKING NO REPRESENTATION THAT THIS
*
* IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
*
* AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
*
* FOR YOUR IMPLEMENTATION. XILINX EXPRESSLY DISCLAIMS ANY
*
* WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
*
* IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
*
* REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
*
* INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
*
* FOR A PARTICULAR PURPOSE.
```

```
*
*
* Xilinx products are not intended for use in life support
*
* appliances, devices, or systems. Use in such applications are
*
* expressly prohibited.
```

```
*
* (c) Copyright 1995-2004 Xilinx, Inc.
```

```
*
* *****/
```

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

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

```
`timescale 1ns/1ps
```

```
module rand(
    clk,
    pd_out,
    ce);
```

```
input clk;
output [7 : 0] pd_out;
input ce;

// synopsys translate_off

    LFSR_V3_0 #(
        "11111111", // c_ainit_val
        0,          // c_enable_rlocs
        0,          // c_gate
        0,          // c_has_ainit
        1,          // c_has_ce
        0,          // c_has_data_valid
        0,          // c_has_load
        0,          // c_has_load_taps
        0,          // c_has_new_seed
        0,          // c_has_pd_in
        1,          // c_has_pd_out
        0,          // c_has_sd_in
        0,          // c_has_sd_out
        0,          // c_has_sinit
        0,          // c_has_taps_in
        0,          // c_has_term_cnt
        0,          // c_implementation
        0,          // c_max_len_logic
        0,          // c_max_len_logic_type
        "11111111", // c_sinit_val
        8,          // c_size
        "00011101", // c_tap_pos
        0)          // c_type
    inst (
        .CLK(clk),
        .PD_OUT(pd_out),
        .CE(ce),
        .SD_OUT(),
        .LOAD(),
        .PD_IN(),
        .SD_IN(),
        .DATA_VALID(),
        .LOAD_TAPS(),
        .TAPS_IN(),
        .SINIT(),
        .AINIT(),
        .NEW_SEED(),
        .TERM_CNT());

// synopsys translate_on

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

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

```
endmodule
```

rand_tb.v

```
`timescale 1ns / 1ps

/////////////////////////////////////////////////////////////////
//////////
// Company:
// Engineer:
//
// Create Date:    17:24:05 05/15/2006
// Design Name:    rand
// Module Name:    rand_tb.v
// Project Name:   ver27
// Target Device:
// Tool versions:
// Description:
//
// Verilog Test Fixture created by ISE for module: rand
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
/////////////////////////////////////////////////////////////////
//////////

module rand_tb_v;

    // Inputs
    reg clk;
    reg ce;

    // Outputs
    wire [7:0] pd_out;

    // Instantiate the Unit Under Test (UUT)
    rand uut (
        .clk(clk),
        .pd_out(pd_out),
        .ce(ce)
    );

    always #10 clk <= ~clk;
    initial begin
        // Initialize Inputs
        clk = 0;
        ce = 0;

        // Wait 100 ns for global reset to finish
        #105;
    end
endmodule
```

```
        // Add stimulus here
        ce = 1;
        #100;
        ce = 0;
        #100;
        ce = 1;
    end
endmodule
```

recorder.v

```
////////////////////////////////////
////////
//
// bi-directional monaural interface to AC97
//
////////////////////////////////////
////////

module audio (clock_27mhz, reset, audio_in_data, audio_out_data, ready,
             audio_reset_b, ac97_sdata_out, ac97_sdata_in,
             ac97_synch, ac97_bit_clock);

    input clock_27mhz;
    input reset;
    output [7:0] audio_in_data;
    input [7:0] audio_out_data;
    output ready;

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

    wire [4:0] volume;
    wire source;
    assign volume = 4'd44; //a reasonable volume value (orig 22)
    assign source = 1; //mic

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

    reg audio_reset_b;
    reg [9:0] reset_count;

    //wait a little before enabling the AC97 codec
    always @(posedge clock_27mhz) begin
        if (reset) begin
```

```
        audio_reset_b = 1'b0;
        reset_count = 0;
    end else if (reset_count == 1023)
        audio_reset_b = 1'b1;
    else
        reset_count = reset_count+1;
    end
end

wire ac97_ready;
ac97 ac97(ac97_ready, command_address, command_data, command_valid,
        left_out_data, 1'b1, right_out_data, 1'b1, left_in_data,
        right_in_data, ac97_sdata_out, ac97_sdata_in, ac97_synch,
        ac97_bit_clock);

// synchronize ready signal with 27Mhz clock
reg ready,ready_pre_sync;
always @ (posedge clock_27mhz) begin
    ready_pre_sync <= ac97_ready;
    ready <= ready_pre_sync;
end

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

assign left_out_data = {audio_out_data, 12'b000000000000};
assign right_out_data = left_out_data;

//arbitrarily choose left input, get highest-order bits
assign audio_in_data = left_in_data[19:12];

endmodule

// assemble/disassemble AC97 serial frames
module ac97 (ready,
        command_address, command_data, command_valid,
        left_data, left_valid,
        right_data, right_valid,
        left_in_data, right_in_data,
        ac97_sdata_out, ac97_sdata_in, ac97_synch, ac97_bit_clock);

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

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

    reg ready;

    reg ac97_sdata_out;
    reg ac97_synch;
```

```
reg [7:0] bit_count;

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

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

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

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

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

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

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

```

end

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

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

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

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

    bit_count <= bit_count+1;

end // always @ (posedge ac97_bit_clock)

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

endmodule

// issue initialization commands to AC97
module ac97commands (clock, ready, command_address, command_data,
                    command_valid, volume, source);

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

```

```
output command_valid;
input [4:0] volume;
input source;

reg [23:0] command;
reg command_valid;

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

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

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

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

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

    case (state)
        4'h0: // Read ID
            begin
                command <= 24'h80_0000;
                command_valid <= 1'b1;
            end
        4'h1: // Read ID
            command <= 24'h80_0000;
        4'h3: // headphone volume
            command <= { 8'h04, 3'b000, vol, 3'b000, vol };
        4'h5: // PCM volume
            command <= 24'h18_0808;
        4'h6: // Record source select
            command <= 24'h1A_0000; // microphone
        4'h7: // Record gain = max
            command <= 24'h1C_0F0F;
        4'h9: // set +20db mic gain
            command <= 24'h0E_8048;
        4'hA: // Set beep volume
            command <= 24'h0A_0000;
        4'hB: // PCM out bypass mix1
            command <= 24'h20_8000;
        default:
    endcase
end
```

```

        command <= 24'h80_0000;
    endcase // case(state)

    old_ready <= ready;

end // always @ (posedge clock)

endmodule // ac97commands

////////////////////////////////////
////////
//
// generate PCM data for 750hz sine wave (assuming f(ready) = 48khz)
//
////////////////////////////////////
////////

module tone750hz (clock, ready, pcm_data);

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

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

    initial begin
        old_ready <= 1'b0;
        // synthesis attribute init of old_ready is "0";
        index <= 8'h00;
        // synthesis attribute init of index is "00";
        pcm_data <= 20'h00000;
        // synthesis attribute init of pcm_data is "00000";
    end

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

    // one cycle of a sinewave in 64 20-bit samples
    always @(index) begin
        case (index[5:0])
            6'h00: pcm_data <= 20'h00000;
            6'h01: pcm_data <= 20'h0C8BD;
            6'h02: pcm_data <= 20'h18F8B;
            6'h03: pcm_data <= 20'h25280;
            6'h04: pcm_data <= 20'h30FBC;
            6'h05: pcm_data <= 20'h3C56B;
            6'h06: pcm_data <= 20'h471CE;
            6'h07: pcm_data <= 20'h5133C;
            6'h08: pcm_data <= 20'h5A827;
            6'h09: pcm_data <= 20'h62F20;
            6'h0A: pcm_data <= 20'h6A6D9;

```

```
6'h0B: pcm_data <= 20'h70E2C;
6'h0C: pcm_data <= 20'h7641A;
6'h0D: pcm_data <= 20'h7A7D0;
6'h0E: pcm_data <= 20'h7D8A5;
6'h0F: pcm_data <= 20'h7F623;
6'h10: pcm_data <= 20'h7FFFF;
6'h11: pcm_data <= 20'h7F623;
6'h12: pcm_data <= 20'h7D8A5;
6'h13: pcm_data <= 20'h7A7D0;
6'h14: pcm_data <= 20'h7641A;
6'h15: pcm_data <= 20'h70E2C;
6'h16: pcm_data <= 20'h6A6D9;
6'h17: pcm_data <= 20'h62F20;
6'h18: pcm_data <= 20'h5A827;
6'h19: pcm_data <= 20'h5133C;
6'h1A: pcm_data <= 20'h471CE;
6'h1B: pcm_data <= 20'h3C56B;
6'h1C: pcm_data <= 20'h30FBC;
6'h1D: pcm_data <= 20'h25280;
6'h1E: pcm_data <= 20'h18F8B;
6'h1F: pcm_data <= 20'h0C8BD;
6'h20: pcm_data <= 20'h00000;
6'h21: pcm_data <= 20'hF3743;
6'h22: pcm_data <= 20'hE7075;
6'h23: pcm_data <= 20'hDAD80;
6'h24: pcm_data <= 20'hCF044;
6'h25: pcm_data <= 20'hC3A95;
6'h26: pcm_data <= 20'hB8E32;
6'h27: pcm_data <= 20'hAECC4;
6'h28: pcm_data <= 20'hA57D9;
6'h29: pcm_data <= 20'h9D0E0;
6'h2A: pcm_data <= 20'h95927;
6'h2B: pcm_data <= 20'h8F1D4;
6'h2C: pcm_data <= 20'h89BE6;
6'h2D: pcm_data <= 20'h85830;
6'h2E: pcm_data <= 20'h8275B;
6'h2F: pcm_data <= 20'h809DD;
6'h30: pcm_data <= 20'h80000;
6'h31: pcm_data <= 20'h809DD;
6'h32: pcm_data <= 20'h8275B;
6'h33: pcm_data <= 20'h85830;
6'h34: pcm_data <= 20'h89BE6;
6'h35: pcm_data <= 20'h8F1D4;
6'h36: pcm_data <= 20'h95927;
6'h37: pcm_data <= 20'h9D0E0;
6'h38: pcm_data <= 20'hA57D9;
6'h39: pcm_data <= 20'hAECC4;
6'h3A: pcm_data <= 20'hB8E32;
6'h3B: pcm_data <= 20'hC3A95;
6'h3C: pcm_data <= 20'hCF044;
6'h3D: pcm_data <= 20'hDAD80;
6'h3E: pcm_data <= 20'hE7075;
6'h3F: pcm_data <= 20'hF3743;
endcase // case(index[5:0])
end // always @ (index)
endmodule
```

```

////////////////////////////////////
//////////
//
// Record/playback
//
////////////////////////////////////
//////////

module recorder(clock_27mhz, reset, playback, ready, from_ac97_data,
to_ac97_data);
    input clock_27mhz;           // 27mhz system clock
    input reset;                // 1 to reset to initial state
    input playback;             // 1 for playback, 0 for record
    input ready;                // 1 when AC97 data is available
    input [7:0] from_ac97_data;  // 8-bit PCM data from mic
    output [7:0] to_ac97_data;   // 8-bit PCM data to headphone

    // detect clock cycle when READY goes 0 -> 1
    // f(READY) = 48khz
    wire new_frame;
    reg old_ready;
    always @ (posedge clock_27mhz) old_ready <= reset ? 0 : ready;
    assign new_frame = ready & ~old_ready;

    // test: playback 750hz tone, or loopback using incoming data
    wire [19:0] tone;
    tone750hz xxx(clock_27mhz, ready, tone);
    reg [7:0] to_ac97_data;
    always @ (posedge clock_27mhz) begin
        if (new_frame) begin
            // just received new data from the AC97
            to_ac97_data <= playback ? tone[19:12] : from_ac97_data;
        end
    end
end

endmodule

```

rect.v

```

module rect(x, y, corner_x, corner_y,
            width, height, color, rgb);
    input [9:0] x; //pixel_count
    input [9:0] y; //line_count;
    input [9:0] corner_x; //pixel count of upper left hand corner
    input [9:0] corner_y; //line count of upper left hand corner
    input [9:0] width;
    input [9:0] height;
    input [23:0] color;
    output [23:0] rgb;

    assign    rgb =
                ( ((x >= corner_x) && (x <= corner_x + width))
                  && ((y >= corner_y) && (y <= corner_y + height)) ) ?
                color : 24'h0;

endmodule // rectangle

```

rotate_dia.v

```
module rotate_dia(old_i, old_j, center_i, center_j, new_i, new_j);
    input [4:0] old_i, center_i;
    input [3:0] old_j, center_j;
    output [4:0] new_i;
    output [3:0] new_j;

    assign new_i = (old_j == center_j - 1) ? (center_i - 1) :
(center_i + 1);
    assign new_j = (old_i == center_i - 1) ? (center_j + 1) :
(center_j - 1);
endmodule
```

rotate_ortho.v

```
rotate_ortho(old_i, old_j, center_i, center_j, new_i, new_j);
    input [4:0] old_i, center_i;
    input [3:0] old_j, center_j;
    output [4:0] new_i;
    output [3:0] new_j;

    assign new_i = ((old_i >= center_i) && (old_j <= center_j)) ?
(old_i - 1) : (old_i + 1);
    assign new_j = ((old_i >= center_i) && (old_j >= center_j)) ?
(old_j - 1) : (old_j + 1);
endmodule
```

scan_sampler.v

```
////////////////////////////////////
//////////
// Module scan_sampler - Processes sync pulses and formats the laser
scan appropriately
////////////////////////////////////
//////////
module scan_sampler(clk, hsync_pulse, vsync_pulse, samples_per_second,
sample_enable, hsync_fps, vsync_fps);
    input clk;
    input hsync_pulse;
    input vsync_pulse;
    input [9:0] samples_per_second;
    input sample_enable;
    output [12:0] hsync_fps;
    output [12:0] vsync_fps;

    reg [9:0] hsync_count = 0;
    reg [9:0] vsync_count = 0;
    reg [12:0] hsync_fps = 0;
    reg [12:0] vsync_fps = 0;

    parameter ONE_SECOND = 26999999;

    always @(posedge clk)
```

```

begin
    if (hsync_pulse) begin
        hsync_count <= hsync_count + 1;
    end
    if (vsync_pulse) begin
        vsync_count <= vsync_count + 1;
    end
    if (sample_enable) begin
        hsync_fps <= hsync_count * samples_per_second;
        vsync_fps <= vsync_count * samples_per_second;
        hsync_count <= 0;
        vsync_count <= 0;
    end
end

end
endmodule

```

self_overlap.v

```

module self_overlap(i0, i1, i2, i3, j0, j1, j2, j3, i, j, result);
    input [4:0] i0, i1, i2, i3, i;
    input [3:0] j0, j1, j2, j3, j;
    output      result;

    assign      result = ( ((i == i0) && (j == j0)) ||
                           ((i == i1) && (j == j1)) ||
                           ((i == i2) && (j == j2)) ||
                           ((i == i3) && (j == j3)) );
endmodule

```

signal_reg.v

```

module signal_reg(reset_sync, signal_reset, signal_sync, signal);
    input reset_sync, signal_reset, signal_sync;
    output signal;

    assign signal = (reset_sync || signal_reset) ? 0 : (signal ? 1 :
signal_sync);
endmodule

```

sum_of_squares.v

```

////////////////////////////////////
//////////
// Module sum_of_squares - Returns the normalized sum of re^2 and im^2
data
////////////////////////////////////
//////////
module sum_of_squares(clk, re, im, sum);

    input clk;
    input [18:0] re;
    input [18:0] im;

```

```

output [37:0] sum;

reg [37:0] sum;

// Switch from 2's complement to positive unsigned int
wire [18:0] re_unsigned = (re[18]) ? ({0,~re[17:0]}+1) : ({0,
re[17:0]});
wire [18:0] im_unsigned = (im[18]) ? ({0,~im[17:0]}+1) : ({0,
im[17:0]});

always @(posedge clk)
begin
    sum <= re_unsigned*re_unsigned + im_unsigned*im_unsigned;
end

endmodule

```

sync_enable.v

```

`timescale 1ns / 1ps
/////////////////////////////////////////////////////////////////
// Module sync_enable - Converts raw sync signals into clean one-cycle
pulses
/////////////////////////////////////////////////////////////////
module sync_enable(clk, hsync_raw, vsync_raw, hsync_pulse, vsync_pulse);
    input clk;
    input hsync_raw;
    input vsync_raw;
    output hsync_pulse;
    output vsync_pulse;

    reg last_hsync = 1;
    reg last_vsync = 1;
    reg hsync_pulse = 0;
    reg vsync_pulse = 0;
    reg [31:0] hsync_smoothing_count = 0;
    reg [31:0] vsync_smoothing_count = 0;

    always @(posedge clk)
    begin
        // Store old sync values
        last_hsync <= hsync_raw;
        last_vsync <= vsync_raw;
        // Disable sync pulse outputs by default
        hsync_pulse <= 0;
        vsync_pulse <= 0;
        if (last_hsync == 1) begin
            hsync_smoothing_count <= hsync_smoothing_count + 1;
        end
        if (last_vsync == 1) begin
            vsync_smoothing_count <= vsync_smoothing_count + 1;
        end
        if (hsync_raw == 0) begin
            if (last_hsync == 1) begin

```

```
100000, then 25000
        if (hsync_smoothing_count > 250) begin // Orig
            // Enable hsync pulse for 1 clock cycle
            hsync_pulse <= 1;
            hsync_smoothing_count <= 0;
            vsync_smoothing_count <= 0;
        end
    end
end
if (vsync_raw == 0) begin
    if (last_vsync == 1) begin
        if (vsync_smoothing_count > 250) begin
            // Enable vsync pulse for 1 clock cycle
            vsync_pulse <= 1;
            hsync_smoothing_count <= 0;
            vsync_smoothing_count <= 0;
        end
    end
end
end
endmodule
```

synchronizer.v

```
`timescale 1ns / 10ps
```

```
module synchronizer (clk, reset, up, down, reset_sync, up_sync,  
down_sync);
```

```
    // Define inputs and outputs  
    input clk, reset, up, down;  
    output reset_sync, up_sync, down_sync;
```

```
    // Instantiate the Reset Debouncer  
    debounce debounce_reset (  
        .reset(reset),  
        .clock(clk),  
        .noisy(reset),  
        .clean(reset_sync)  
    );
```

```
    // Instantiate the Up Debouncer  
    debounce debounce_up (  
        .reset(reset_sync),  
        .clock(clk),  
        .noisy(up),  
        .clean(up_sync)  
    );
```

```
    // Instantiate the Down Debouncer  
    debounce debounce_down (  
        .reset(reset_sync),  
        .clock(clk),  
        .noisy(down),  
        .clean(down_sync)  
    );
```

```
endmodule
```

vga.v

```
// This module provides control signals to the ADV7125.
// The resolution is 640x480 and the pixel frequency
// is about 25MHz
// hsync is active low: high for 640 pixels of active video,
// high for 16 pixels of front porch,
// low for 96 pixels of hsync,
// high for 48 pixels of back porch
// vsync is active low: high for 480 lines of active video,
// high for 11 lines of front porch,
// low for 2 lines of vsync,
// high for 32 lines of back porch

module vga (pixel_clock, reset, hsync, vsync, sync_b,
           blank_b, pixel_count, line_count, update_frame);
  input pixel_clock; // 31.5 MHz pixel clock
  input reset; // system reset
  output hsync; // horizontal sync
  output vsync; // vertical sync
  output sync_b; // hardwired to Vdd
  output blank_b; // composite blank
  output [9:0] pixel_count; // number of the current pixel
  output [9:0] line_count; // number of the current line
  output update_frame;

  // 640x480 75Hz parameters
  parameter PIXELS = 800;
  parameter LINES = 525;
  parameter HACTIVE_VIDEO = 640;
  parameter HFRONT_PORCH = 16;
  parameter HSYNC_PERIOD = 96;
  parameter HBACK_PORCH = 48;
  parameter VACTIVE_VIDEO = 480;
  parameter VFRONT_PORCH = 11;
  parameter VSYNC_PERIOD = 2;
  parameter VBACK_PORCH = 32;

  // current pixel count
  reg [9:0] pixel_count = 10'b0;
  reg [9:0] line_count = 10'b0;

  // registered outputs
  reg hsync = 1'b1;
  reg vsync = 1'b1;
  reg blank_b = 1'b1;
  wire sync_b; // connected to Vdd
  wire pixel_clock;
  wire [9:0] next_pixel_count;
  wire [9:0] next_line_count;

  reg old_vsync;
  assign update_frame = (!vsync && old_vsync);

  always @ (posedge pixel_clock)
```

```
begin
  old_vsync <= vsync;

  if (reset)
    begin
      pixel_count <= 10'b0;
      line_count <= 10'b0;
      hsync <= 1'b1;
      vsync <= 1'b1;
      blank_b <= 1'b1;
    end
  else
    begin
      pixel_count <= next_pixel_count;
      line_count <= next_line_count;
      hsync <=
        (next_pixel_count < HACTIVE_VIDEO + HFRONT_PORCH) |
        (next_pixel_count >= HACTIVE_VIDEO+HFRONT_PORCH+
         HSYNC_PERIOD);
      vsync <=
        (next_line_count < VACTIVE_VIDEO+VFRONT_PORCH) |
        (next_line_count >= VACTIVE_VIDEO+VFRONT_PORCH+
         VSYNC_PERIOD);
      // this is the and of hblank and vblank
      blank_b <=
        (next_pixel_count < HACTIVE_VIDEO) &
        (next_line_count < VACTIVE_VIDEO);
    end
  end
  // next state is computed with combinational logic
  assign next_pixel_count = (pixel_count == PIXELS-1) ?
    10'h000 : pixel_count + 1'b1;
  assign next_line_count = (pixel_count == PIXELS-1) ?
    (line_count == LINES-1) ? 10'h000 :
    line_count + 1'b1 : line_count;

  // since we are providing hsync and vsync to the display, we
  // can hardwire composite sync to Vdd.
  assign sync_b = 1'b1;
endmodule
```