```
//////////////////////////////////////////////////////////////////////////////
//
// 6.111 FPGA Labkit -- Template Toplevel Module
//
// For Labkit Revision 004
//
//
// Created: October 31, 2004, from revision 003 file
// Author: Nathan Ickes
//
//////////////////////////////////////////////////////////////////////////////
//
// CHANGES FOR BOARD REVISION 004
//
// 1) Added signals for logic analyzer pods 2-4.
// 2) Expanded "tv_in_ycrcb" to 20 bits.
// 3) Renamed "tv_out_data" to "tv_out_i2c_data" and "tv_out_sclk" to
//    "tv_out_i2c_clock".
// 4) Reversed disp_data_in and disp_data_out signals, so that "out" is an
//    output of the FPGA, and "in" is an input.
//
// CHANGES FOR BOARD REVISION 003
//
// 1) Combined flash chip enables into a single signal, flash_ce_b.
//
// CHANGES FOR BOARD REVISION 002
//
// 1) Added SRAM clock feedback path input and output
// 2) Renamed "mousedata" to "mouse_data"
// 3) Renamed some ZBT memory signals. Parity bits are now incorporated into
//    the data bus, and the byte write enables have been combined into the
//    4-bit ram#_bwe_b bus.
// 4) Removed the "systemace_clock" net, since the SystemACE clock is now
//    hardwired on the PCB to the oscillator.
//
//////////////////////////////////////////////////////////////////////////////
//
// Complete change history (including bug fixes)
//
// 2005-Jan-23: Reduced flash address bus to 24 bits, to match 128Mb devices
//              actually populated on the boards. (The boards support up to
//              256Mb devices, with 25 address lines.)
//
// 2004-Apr-29: Change history started
//
// 2004-Apr-29: Reduced SRAM address busses to 19 bits, to match 18Mb devices
//              actually populated on the boards. (The boards support up to
//              72Mb devices, with 21 address lines.)
//
// 2004-May-01: Changed "disp_data_in" to be an output, and gave it a default
//              value. (Previous versions of this file declared this port to
//              be an input.)
//
// 2004-Oct-31: Adapted to new revision 004 board.
//
//////////////////////////////////////////////////////////////////////////////

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

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

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

               tv_in_ycrcb, tv_in_data_valid, tv_in_line_clock1,
               tv_in_line_clock2, tv_in_aef, tv_in_hff, tv_in_aff,
               tv_in_i2c_clock, tv_in_i2c_data, tv_in_fifo_read,
               tv_in_fifo_clock, tv_in_iso, tv_in_reset_b, tv_in_clock,

               ram0_data, ram0_address, ram0_adv_ld, ram0_clk, ram0_cen_b,
               ram0_ce_b, ram0_oe_b, ram0_we_b, ram0_bwe_b,

               ram1_data, ram1_address, ram1_adv_ld, ram1_clk, ram1_cen_b,
               ram1_ce_b, ram1_oe_b, ram1_we_b, ram1_bwe_b,

               clock_feedback_out, clock_feedback_in,

               flash_data, flash_address, flash_ce_b, flash_oe_b, flash_we_b,
               flash_reset_b, flash_sts, flash_byte_b,

               rs232_txd, rs232_rxd, rs232_rts, rs232_cts,

               mouse_clock, mouse_data, keyboard_clock, keyboard_data,

               clock_27mhz, clock1, clock2,

               disp_blank, disp_data_out, disp_clock, disp_rs, disp_ce_b,
               disp_reset_b, disp_data_in,

               button0, button1, button2, button3, button_enter, button_right,
               button_left, button_down, button_up,

               switch,

               led,

               user1, user2, user3, user4,

               daughtercard,

               systemace_data, systemace_address, systemace_ce_b,
               systemace_we_b, systemace_oe_b, systemace_irq, systemace_mpbrdy,

               analyzer1_data, analyzer1_clock,
               analyzer2_data, analyzer2_clock,
               analyzer3_data, analyzer3_clock,
               analyzer4_data, analyzer4_clock^M
^M
                  );

   output beep, audio_reset_b, ac97_synch, ac97_sdata_out;
   input  ac97_bit_clock, ac97_sdata_in;

   output [7:0] vga_out_red, vga_out_green, vga_out_blue;
   output vga_out_sync_b, vga_out_blank_b, vga_out_pixel_clock,
          vga_out_hsync, vga_out_vsync;

   output [9:0] tv_out_ycrcb;
   output tv_out_reset_b, tv_out_clock, tv_out_i2c_clock, tv_out_i2c_data,
          tv_out_pal_ntsc, tv_out_hsync_b, tv_out_vsync_b, tv_out_blank_b,
          tv_out_subcar_reset;

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

```
    inout  tv_in_i2c_data;

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

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

    input  clock_feedback_in;
    output clock_feedback_out;

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

    output rs232_txd, rs232_rts;
    input  rs232_rxd, rs232_cts;

    input  mouse_clock, mouse_data, keyboard_clock, keyboard_data;

    input  clock_27mhz, clock1, clock2;

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

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

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

    inout [43:0] daughtercard;

    inout  [15:0] systemace_data;
    output [6:0]  systemace_address;
    output systemace_ce_b, systemace_we_b, systemace_oe_b;
    input  systemace_irq, systemace_mpbrdy;

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

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

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

    // VGA Output^M
    // disable, outputting below^M
    /*
    assign vga_out_red = 10'h0;
    assign vga_out_green = 10'h0;
    assign vga_out_blue = 10'h0;
    assign vga_out_sync_b = 1'b1;
    assign vga_out_blank_b = 1'b1;
    assign vga_out_pixel_clock = 1'b0;
    assign vga_out_hsync = 1'b0;
    assign vga_out_vsync = 1'b0;^M
    */

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

    // Video Input
    assign tv_in_i2c_clock = 1'b0;
    assign tv_in_fifo_read = 1'b0;
    assign tv_in_fifo_clock = 1'b0;
    assign tv_in_iso = 1'b0;
    assign tv_in_reset_b = 1'b0;
    assign tv_in_clock = 1'b0;
    assign tv_in_i2c_data = 1'bZ;
    // tv_in_ycrcb, tv_in_data_valid, tv_in_line_clock1, tv_in_line_clock2,
    // tv_in_aef, tv_in_hff, and tv_in_aff are inputs

    // SRAMs
    assign ram0_data = 36'hZ;
    assign ram0_address = 19'h0;
    assign ram0_adv_ld = 1'b0;
    assign ram0_clk = 1'b0;
    assign ram0_cen_b = 1'b1;
    assign ram0_ce_b = 1'b1;
    assign ram0_oe_b = 1'b1;
    assign ram0_we_b = 1'b1;
    assign ram0_bwe_b = 4'hF;
    assign ram1_data = 36'hZ;
    assign ram1_address = 19'h0;
    assign ram1_adv_ld = 1'b0;
    assign ram1_clk = 1'b0;
    assign ram1_cen_b = 1'b1;
    assign ram1_ce_b = 1'b1;
    assign ram1_oe_b = 1'b1;
    assign ram1_we_b = 1'b1;
    assign ram1_bwe_b = 4'hF;
    assign clock_feedback_out = 1'b0;
    // clock_feedback_in is an input

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

    // RS-232 Interface
```

```
   // assign rs232_txd = 1'b1;
   // assign rs232_rts = 1'b1;
   // rs232_rxd and rs232_cts are inputs

   // PS/2 Ports
   // mouse_clock, mouse_data, keyboard_clock, and keyboard_data are inputs

   // LED Displays
   assign disp_blank = 1'b1;
   assign disp_clock = 1'b0;
   assign disp_rs = 1'b0;
   assign disp_ce_b = 1'b1;
   assign disp_reset_b = 1'b0;
   // disp_data_out is an input

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

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

   // User I/Os
   assign user1 = 32'hZ;
   assign user2 = 32'hZ;
   assign user3 = 32'hZ;
   assign user4 = 32'hZ;

   // Daughtercard Connectors
   assign daughtercard = 44'hZ;

   // SystemACE Microprocessor Port
   assign systemace_data = 16'hZ;
   assign systemace_address = 7'h0;
   assign systemace_ce_b = 1'b1;
   assign systemace_we_b = 1'b1;
   assign systemace_oe_b = 1'b1;
   // systemace_irq and systemace_mpbrdy are inputs

        wire finished;
        wire [11:0] master_rom_add, rom_add, video_rom_add;
        wire [63:0] rom_out, col_ram_in;
        wire [63:0] row_ram_out, col_ram_out;
        wire [19:0] results_ram_in;
        wire [19:0] results_ram_out;
        wire [5:0] row_ram_add, col_ram_add, results_ram_add;
        wire [5:0] master_results_ram_add, video_results_ram_add;
        wire row_ram_we, col_ram_we, results_ram_we;

        // pixel clock
        pixel_clock pc(.CLKIN_IN(clock_27mhz),
                       .CLKFX_OUT(vga_out_pixel_clock));


        // synchronizer
        wire reset_sync, export_spice_sync;
        wire [1:0] control_sync;
        wire [1:0] master_control;
        assign master_control = finished ? control_sync : 2'b11;

        synchronizer s(.clk(clock_27mhz),
                       .reset(~button0),
                       .control(switch[1:0]),
                       .export_spice(~button1),
                       .reset_sync(reset_sync),
                       .control_sync(control_sync),
                       .export_spice_sync(export_spice_sync));

        // video top
        video_top vt(.clk(vga_out_pixel_clock),
                     .reset(reset_sync),
                     .switches(master_control),
                     .ideal_data(results_ram_out),
                     .raw_ckt_dout(rom_out),
                     .vga_out_hsync(vga_out_hsync),
                     .vga_out_vsync(vga_out_vsync),
                     .vga_out_blank_b(vga_out_blank_b),
                     .vga_out_sync_b(vga_out_sync_b),
                     .vga_out_red(vga_out_red),
                     .vga_out_green(vga_out_green),
                     .vga_out_blue(vga_out_blue),
                     .ideal_addr(video_results_ram_add),
                     .raw_ckt_addr(video_rom_add),
                     .serial_export(export_spice_sync),
                     .rs232_rts(rs232_rts),
                     .rs232_txd(rs232_txd)
                     );


        // rom_64x4096 and resultsram_20x64 are shared between ravi's and vijay's modules
        wire shared_mem_clk;
        assign shared_mem_clk = finished ? vga_out_pixel_clock : clock_27mhz;
        assign master_rom_add = finished ? video_rom_add : rom_add;
        assign master_results_ram_add = finished ? video_results_ram_add : results_ram_add;

        rom_64x4096 my_image_rom(.addr(master_rom_add), .clk(shared_mem_clk), .dout(rom_out));
        rowram_64x64 my_row_ram(.addr(row_ram_add), .clk(clock_27mhz),
                                .din(rom_out), .dout(row_ram_out), .we(row_ram_we));
        colram_64x64 my_col_ram(.addr(col_ram_add), .clk(clock_27mhz),
                                .din(col_ram_in), .dout(col_ram_out), .we(col_ram_we));

        resultsram_20x64 my_results_ram(.addr(master_results_ram_add), .clk(shared_mem_clk),
                                .din(results_ram_in), .dout(results_ram_out), .we(results_ram_we));

        wire [79:0] val_reg;
    wire [9:0] val_pads;
    wire [6:0] mult_pads;

        ravi_fsm my_ravi_fsm(.clk(clock_27mhz), .reset_sync(reset_sync), .finished(finished),
                     .rom_add(rom_add),
                                .row_ram_add(row_ram_add), .row_ram_out(row_ram_out), .row_ram_we(row_ram_we),
                                .col_ram_add(col_ram_add), .col_ram_in(col_ram_in), .col_ram_out(col_ram_out),
                                .col_ram_we(col_ram_we),
                                .results_ram_add(results_ram_add), .results_ram_in(results_ram_in),
                                .results_ram_we(results_ram_we), .val_reg(val_reg), .val_pads(val_pads), .mult_pads(mult_pads));

            // Logic Analyzer


   assign analyzer1_data = 16'h0;
   assign analyzer1_clock = 1'b1;
   assign analyzer2_data = 16'h0;
   assign analyzer2_clock = 1'b1;
   assign analyzer3_data = 16'h0;
   assign analyzer3_clock = 1'b1;
   assign analyzer4_data = 16'h0;
   assign analyzer4_clock = 1'b1;
endmodule
```

```
module synchronizer (clk,
                     reset, control, export_spice,
                     reset_sync, control_sync, export_spice_sync);
   input clk;
   input reset;
   input [1:0] control;
   input       export_spice;

   output      reset_sync;
   output [1:0] control_sync;
   output       export_spice_sync;


   reg          reset_int;
   reg [1:0]    control_int;
   reg          export_spice_int;
   reg          reset_sync;
   reg [1:0]    control_sync;
   reg          export_spice_sync;


   always @ (posedge clk) begin
      // input signals will each go through a chain of 2 flipflops
      // to produce synchronized signals unlikely to be metastable
      // use nonblocking assignment
      reset_int <= reset;
      control_int <= control;
      export_spice_int <= export_spice;

      reset_sync <= reset_int;
      control_sync <= control_int;
      export_spice_sync <= export_spice_int;
   end
endmodule // synchronizer
```

```
1   module txt_fsm(clk,reset_sync,start,finished,
2                  row_ram_add, row_ram_out,
3                  col_ram_add, col_ram_out,
4                  val, mult, val_reg, val_pads, mult_pads);
5       input clk, reset_sync, start;
6       output finished;
7       reg finished;
8       output [5:0] row_ram_add, col_ram_add;
9       reg [5:0] row_ram_add, col_ram_add;
10      input [63:0] row_ram_out, col_ram_out;
11      output [11:0] val;
12      reg [11:0] val;
13      output [2:0] mult;
14      reg [2:0] mult;
15      reg [5:0] bot, right;
16
17      output [79:0] val_reg;
18      reg [79:0] val_reg;
19      reg [2:0] col_counter;
20      reg [1:0] digit_counter;
21
22      wire one, two, three, four, five, six, seven, eight, nine, zero;
23      wire femto, pico, nano, micro, milli, kilo, mega;
24
25      output [9:0] val_pads;
26      output [6:0] mult_pads;
27
28      assign val_pads = {zero, one, two, three, four, five, six, seven, eight, nine};
29      assign mult_pads = {femto, pico, nano, micro, milli, kilo, mega};
30
31
32  assign one =   (!val_reg[77] && !val_reg[76] && !val_reg[75] &&
33                  !val_reg[67] && !val_reg[66] && !val_reg[65] &&
34                  !val_reg[57] && !val_reg[56] && !val_reg[55]) &&
35                 (!val_reg[50] && !val_reg[40] && !val_reg[30] &&
36                 !val_reg[51] && !val_reg[41] && !val_reg[31]) &&
37                 (!val_reg[59] && !val_reg[49] && !val_reg[39] &&
38                 !val_reg[58] && !val_reg[48] && !val_reg[38]) &&
39                 (val_reg[16] || val_reg[15] || val_reg[14] ||
40                 val_reg[6] || val_reg[5] || val_reg[4]);
41
42
43  assign two =   (!val_reg[75] && !val_reg[65] && !val_reg[56]) &&
44                 (!val_reg[13] && !val_reg[12] && !val_reg[3] && !val_reg[2]) &&
45                 (val_reg[50] || val_reg[61] || val_reg[72]) &&
46                 (val_reg[27] || val_reg[17] || val_reg[7]);
47
48
49  assign three = (val_reg[59] || val_reg[49] || val_reg[39] ||
50                  val_reg[58] || val_reg[48] || val_reg[38]) &&
51                 ((!val_reg[77] || !val_reg[76]) && (!val_reg[67] || !val_reg[66]) &&
52                 (!val_reg[57] || !val_reg[56])) &&
53                 ((!val_reg[74] || !val_reg[73]) && (!val_reg[64] || !val_reg[63]) &&
54                 (!val_reg[54] || !val_reg[53])) &&
55                 (val_reg[46] || val_reg[45] || val_reg[44] ||
56                  val_reg[36] || val_reg[35] || val_reg[34]) &&
57                 (val_reg[13] || val_reg[12] || val_reg[3] || val_reg[2]) &&
58                 (val_reg[17] || val_reg[16] || val_reg[7] || val_reg[6]);
59
60
61  assign four =  (!val_reg[49] && !val_reg[39] && !val_reg[48] && !val_reg[38]) &&
62                 (val_reg[46] || val_reg[45] || val_reg[43]) &&
63                 (!val_reg[40] && !val_reg[30] && !val_reg[41] && !val_reg[31]);
```

```
64
65
66
67          assign five     = (   ((!val_reg[18] || !val_reg[17]) && (!val_reg[8] || !val_reg[7]))
     ||
68                                ((!val_reg[17] || !val_reg[16]) && (!val_reg[7] || !val_reg[6]))
     ) && //pad1
69                            (!val_reg[73] && !val_reg[63] && !val_reg[53]) &&
70                            (val_reg[76] || val_reg[66] || val_reg[56]) &&
71                            (val_reg[77] || val_reg[67] || val_reg[57]) &&
72                            (val_reg[47] || val_reg[46] || val_reg[45] || val_reg[44] ||
73                             val_reg[37] || val_reg[36] || val_reg[35] || val_reg[34]); //pad2
74
75          assign six      = ((!val_reg[17] || !val_reg[16]) && (!val_reg[7] || !val_reg[6])) &&
76                            ((val_reg[76] && val_reg[75]) || (val_reg[66] && val_reg[65]) || (va
    l_reg[56] && val_reg[55])) &&
77                            (val_reg[73] || val_reg[63] || val_reg[53]) &&
78                            (val_reg[45] || val_reg[44] || val_reg[43] ||
79                             val_reg[35] || val_reg[34] || val_reg[33]); //pad2
80
81          assign seven    = (!val_reg[76] && !val_reg[66] && !val_reg[56]) && //pad1
82                            (val_reg[49] || val_reg[48]) && //pad2
83                            (!val_reg[61] && !val_reg[72] && !val_reg[50]) &&
84                            (val_reg[27] || val_reg[17] || val_reg[7]) &&
85                            (!val_reg[73]  && !val_reg[63]); //pad3
86
87
88          assign eight    = (val_reg[46] || val_reg[45] || val_reg[44] || val_reg[43] ||
89                             val_reg[36] || val_reg[35] || val_reg[34] || val_reg[33]) &&
90                            ( val_reg[73] || val_reg[63] || val_reg[72] || val_reg[62] || val_re
    g[53] || val_reg[52] )&&
91                            (val_reg[16] || val_reg[6] || val_reg[26]) &&
92                            (val_reg[17] || val_reg[7] || val_reg[27]) &&
93                            (val_reg[76] || val_reg[66] || val_reg[56]) &&
94                            (val_reg[13] || val_reg[3] || val_reg[23]);
95
96          assign nine =  ((val_reg[17] && val_reg[16]) || (val_reg[7] && val_reg[16])) &&
97                    (val_reg[49] || val_reg[39] ||
98                     val_reg[48] || val_reg[38]) &&
99                    (!val_reg[51] && !val_reg[41] && !val_reg[31] &&
100                    !val_reg[50] && !val_reg[40] && !val_reg[30]) &&
101                   (val_reg[77] || val_reg[76] || val_reg[67] || val_reg[66]) &&
102                   (val_reg[46] || val_reg[45] || val_reg[44]) &&
103                   (!val_reg[73] && !val_reg[72] &&
104                    !val_reg[63] && !val_reg[62] &&
105                    !val_reg[53] && !val_reg[52]);
106
107         assign zero =  (!val_reg[46] && !val_reg[45] && !val_reg[44] && !val_reg[43] &&
108                         !val_reg[36] && !val_reg[35] && !val_reg[34] && !val_reg[33])  &&
109                        (val_reg[77] || val_reg[76] || val_reg[73] || val_reg[72] ||
110                         val_reg[67] || val_reg[66] || val_reg[63] || val_reg[62]) &&
111                        (val_reg[17] || val_reg[16] || val_reg[13] || val_reg[12] ||
112                         val_reg[7] || val_reg[6] || val_reg[3] || val_reg[2]);
113
114
115
116         assign femto    = ( (!val_reg[27] || !val_reg[26]) &&
117                               (!val_reg[17] || !val_reg[16]) && (!val_reg[7] || !val_reg[6])) &
    &
118                           (val_reg[49] || val_reg[48]) &&
119                           (!val_reg[32] && !val_reg[33] &&
120                            !val_reg[42] && !val_reg[43]);
121
```

```
122     assign pico     = (val_reg[49] || val_reg[48]) &&
123                       ((val_reg[27] && val_reg[26]) ||
124                       (val_reg[17] && val_reg[16]) || (val_reg[7] && val_reg[6])) &&
125                       (!val_reg[32] && !val_reg[33] &&
126                        !val_reg[42] && !val_reg[43]);
127
128     assign nano     = (val_reg[16] || val_reg[15] || val_reg[6] || val_reg[5]) &&
129                       (!val_reg [50] && !val_reg[40] && !val_reg[30]) &&
130                       (!val_reg[49] && !val_reg[48] &&
131                        !val_reg[39] && !val_reg[38]) &&
132                       (val_reg[35] || val_reg[34] || val_reg[33] ||
133                        val_reg[45] || val_reg[44] || val_reg[43]) &&
134                       ((val_reg[21] && val_reg[10])||
135                        (val_reg[22] && val_reg[11])||
136                        (val_reg[23] && val_reg[12]));
137
138
139
140   assign micro =      (val_reg[77] || val_reg[76] || val_reg[67] || val_reg[66]) &&
141             (val_reg[7] || val_reg[6] || val_reg[17] || val_reg[16]) &&
142             (val_reg[4] || val_reg[3] || val_reg[2] ||
143              val_reg[14] || val_reg[13] || val_reg[12]) &&
144             (val_reg[50] || val_reg[40] || val_reg[30] ||
145              val_reg[51] || val_reg[41] || val_reg[31]) &&
146             (!val_reg[59] && !val_reg[49] && !val_reg[39] &&
147              !val_reg[58] && !val_reg[48] && !val_reg[38]) &&
148             (!val_reg[55] && !val_reg[54] && !val_reg[53] &&
149              !val_reg[45] && !val_reg[44] && !val_reg[43] &&
150             !val_reg[35] && !val_reg[34] && !val_reg[33]);
151
152
153   assign milli =      (!val_reg[68] && !val_reg[77] && !val_reg[76] && !val_reg[67] &&
154                        !val_reg[66]) &&
155                       ((val_reg[73] && val_reg[72]) || (val_reg[63] && val_reg[62]));
156
157
158
159     assign kilo     = ((val_reg[77] && val_reg[76]) || (val_reg[67] && val_reg[66])) &&
160                       ((!val_reg[50] || !val_reg[40]) && (!val_reg[51] || !val_reg[41])) &
    &
161                       ((!val_reg[15] || !val_reg[14]) && (!val_reg[5] || !val_reg[4])) &&
162                       ((!val_reg[59] || !val_reg[49]) && (!val_reg[58] || !val_reg[48]));
163
164     assign mega     = ((val_reg[77] && val_reg[76]) || (val_reg[67] && val_reg[66])) &&
165                       (val_reg[56] || val_reg[55] || val_reg[46] || val_reg[45] || val_reg
    [36] || val_reg[35]) &&
166                       (val_reg[17] || val_reg[16] || val_reg[7] || val_reg[6]) &&
167                       ((!val_reg[21] || !val_reg[10])&&
168                        (!val_reg[22] || !val_reg[11])&&
169                        (!val_reg[23] || !val_reg[12])) &&
170                       (!val_reg[39] && !val_reg[38] && !val_reg[49] && !val_reg[48]);
171
172
173     reg [5:0] state;
174     parameter idle = 0;
175     parameter delay0 = 1;
176     parameter find_right_edge0 = 2;
177     parameter delay1 = 3;
178     parameter find_right_edge1 = 4;
179     parameter find_bot_edge0 = 5;
180     parameter delay2 = 6;
181     parameter find_bot_edge1 = 7;
182     parameter delay3 = 8;
```

```
183     parameter fill_val_reg = 9;
184     parameter delay4 = 10;
185     parameter analyze_val = 11;
186     parameter delay5 = 12;
187     parameter fill_val_reg_mult = 13;
188     parameter delay6 = 14;
189     parameter analyze_mult = 15;
190     parameter done = 31;
191
192
193
194     always @ (posedge clk) begin
195     case (state)
196
197      idle: if (start) state <= delay0;
198      else begin
199           state <= idle;
200           col_ram_add <= 6'd63;
201           row_ram_add <= 6'd63;
202           val_reg <= 80'b0;
203           finished <= 0;
204           bot <= 6'd60;
205           right <= 6'd60;
206           col_counter <= 0;
207           digit_counter <= 0;
208      end
209
210      //trying to fix up alignment on right and bottom
211      delay0: state <= find_right_edge0;
212
213      find_right_edge0:
214      if (&col_ram_out[28:21]) begin
215           state <= delay1;
216           col_ram_add <= 6'd62;
217      end
218      else begin
219           state <= find_bot_edge0;
220           right <= 6'd61;
221      end
222
223      delay1: state <= find_right_edge1;
224
225      find_right_edge1: begin
226           if (&col_ram_out[28:21]) right <= 6'd59;
227           else right <= 6'd60;
228           state <= find_bot_edge0;
229      end
230
231      find_bot_edge0: begin
232      col_ram_add <= right - 27;
233      if (&row_ram_out[32:29]) begin
234           state <= delay2;
235           row_ram_add <= 6'd62;
236      end
237      else begin
238           state <= delay3;
239           bot <= 2;
240      end
241      end
242
243      delay2: state <= find_bot_edge1;
244
245      find_bot_edge1: begin
```

```
246             if (&row_ram_out[32:25]) bot <= 4;
247             else bot <= 3;
248             state <= delay3;
249         end
250
251     delay3: state <= fill_val_reg;
252
253     fill_val_reg: begin
254     val_reg <= {val_reg[69:0], {col_ram_out[bot+21], col_ram_out[bot+20],
255                                 col_ram_out[bot+19], col_ram_out[bot+18],
256                                 col_ram_out[bot+17], col_ram_out[bot+16],
257                                 col_ram_out[bot+15], col_ram_out[bot+14],
258                                 col_ram_out[bot+13], col_ram_out[bot+12]}};
259     col_ram_add <= col_ram_add + 1;
260     if (&col_counter) begin
261     col_counter <= 0;
262     col_ram_add <= col_ram_add + 3;
263     state <= delay4;
264     end
265     else begin
266     state <= delay3;
267     col_ram_add <= col_ram_add + 1;
268     col_counter <= col_counter + 1;
269     end
270     end
271
272     delay4: state <= analyze_val;
273
274     analyze_val: begin
275     case (val_pads)
276     10'b1000000000: val <= {val[7:0], 4'd0};
277     10'b0100000000: val <= {val[7:0], 4'd1};
278     10'b0010000000: val <= {val[7:0], 4'd2};
279     10'b0001000000: val <= {val[7:0], 4'd3};
280     10'b0000100000: val <= {val[7:0], 4'd4};
281     10'b0000010000: val <= {val[7:0], 4'd5};
282     10'b0000001000: val <= {val[7:0], 4'd6};
283     10'b0000000100: val <= {val[7:0], 4'd7};
284     10'b0000000010: val <= {val[7:0], 4'd8};
285     10'b0000000001: val <= {val[7:0], 4'd9};
286     10'b0001000010: val <= {val[7:0], 4'd8}; //8 takes priority over 3
287     10'b0000101000: val <= {val[7:0], 4'd6}; //6 takes priority over 4
288     default: val <= {val[7:0], 4'b1111};
289     endcase
290     if (digit_counter[1]) begin
291     col_ram_add <= col_ram_add - 10;
292     state <= delay5;
293     end
294     else begin
295     state <= delay3;
296     digit_counter <= digit_counter + 1;
297     end
298     end
299
300     delay5: state <= fill_val_reg_mult;
301
302     fill_val_reg_mult: begin
303     val_reg <= {val_reg[69:0], {col_ram_out[bot+9], col_ram_out[bot+8],
304                                 col_ram_out[bot+7], col_ram_out[bot+6],
305                                 col_ram_out[bot+5], col_ram_out[bot+4],
306                                 col_ram_out[bot+3], col_ram_out[bot+2],
307                                 col_ram_out[bot+1], col_ram_out[bot]}};
308     col_ram_add <= col_ram_add + 1;
```

```
309         if (&col_counter) begin
310         col_counter <= 0;
311         state <= delay6;
312         end
313         else begin
314         state <= delay5;
315         col_ram_add <= col_ram_add + 1;
316         col_counter <= col_counter + 1;
317         end
318         end
319
320         delay6: state <= analyze_mult;
321
322         analyze_mult:  begin
323         case (mult_pads)
324         7'b1000000: mult <= 3'd0;
325         7'b0100000: mult <= 3'd1;
326         7'b0010000: mult <= 3'd2;
327         7'b0001000: mult <= 3'd3;
328         7'b0000100: mult <= 3'd4;
329         7'b0000010: mult <= 3'd5;
330         7'b0000001: mult <= 3'd6;
331         default: mult <= 3'd7;
332         endcase
333         state <= done;
334         end
335
336
337         done: begin
338         state <= idle;
339         finished <= 1;
340         end
341
342      endcase
343
344      if (reset_sync) begin
345          state <= idle;
346          col_ram_add <= 6'd63;
347          row_ram_add <= 6'd63;
348          val_reg <= 80'b0;
349          finished <= 0;
350          bot <= 6'd60;
351          right <= 6'd60;
352          col_counter <= 0;
353          digit_counter <= 0;
354          val <= 12'd0;
355          mult <= 3'd0;
356      end
357      end
358
359   endmodule
```

```
1   module adder_10x6(row0,row1,row2,row3,row4,row5,sum);
2       input [9:0] row0;
3       input [9:0] row1;
4       input [9:0] row2;
5       input [9:0] row3;
6       input [9:0] row4;
7       input [9:0] row5;
8       output [5:0] sum;
9
10      assign sum = row0[0]+row0[1]+row0[2]+row0[3]+row0[4]+row0[5]+
11                   row0[6]+row0[7]+row0[8]+row0[9]+
12                   row1[0]+row1[1]+row1[2]+row1[3]+row1[4]+row1[5]+
13                   row1[6]+row1[7]+row1[8]+row1[9]+
14                   row2[0]+row2[1]+row2[2]+row2[3]+row2[4]+row2[5]+
15                   row2[6]+row2[7]+row2[8]+row2[9]+
16                   row3[0]+row3[1]+row3[2]+row3[3]+row3[4]+row3[5]+
17                   row3[6]+row3[7]+row3[8]+row3[9]+
18                   row4[0]+row4[1]+row4[2]+row4[3]+row4[4]+row4[5]+
19                   row4[6]+row4[7]+row4[8]+row4[9]+
20                   row5[0]+row5[1]+row5[2]+row5[3]+row5[4]+row5[5]+
21                   row5[6]+row5[7]+row5[8]+row5[9];
22
23
24  //60 input 1 bit adder
25
26  endmodule
27
```

```
1   module choose_fsm(clk, reset_sync,
2                     start, finished,
3                     row_ram_add, col_ram_add,
4                     type,
5                     row_ram_out, col_ram_out);
6
7
8
9   input clk, reset_sync, start;
10  output finished;
11  output [5:0] row_ram_add, col_ram_add;
12  output [4:0] type;
13  wire [5:0] sum; //not used
14
15  wire t1_finished, v2_finished, h2_finished, ltb3_finished, rtb3_finished; //minor fsms fin
    ished?
16  reg t1_start, v2_start, h2_start, ltb3_start, rtb3_start; //minor fsms start
17  wire [5:0] v2_row_ram_add;
18  wire [5:0] h2_col_ram_add;
19  wire [5:0] ltb3_row_ram_add, ltb3_col_ram_add;
20  wire [5:0] rtb3_row_ram_add, rtb3_col_ram_add;
21  wire [5:0] t1_row_ram_add;
22  wire [4:0] t1_type, v2_type, h2_type, ltb3_type, rtb3_type;
23  wire [35:0] tester;
24  reg [2:0] s; //control who controls RAMs
25
26  reg [5:0] top_row_ram_add, top_col_ram_add;
27  reg [4:0] top_type;
28  input [63:0] row_ram_out, col_ram_out;
29  reg [3:0] edges;
30
31  assign tester = row_ram_out[60:25];
32
33  assign row_ram_add = s[2] ? (s[1] ? (s[0] ? top_row_ram_add : top_row_ram_add) :
34                                      (s[0] ? rtb3_row_ram_add : ltb3_row_ram_add)) :
35                              (s[1] ? (s[0] ? top_row_ram_add  : v2_row_ram_add) :
36                                      (s[0]?  t1_row_ram_add :  top_row_ram_add));
37
38  assign col_ram_add = s[2] ? (s[1] ? (s[0] ? top_col_ram_add : top_col_ram_add) :
39                                      (s[0] ? rtb3_col_ram_add : ltb3_col_ram_add)) :
40                              (s[1] ? (s[0] ? h2_col_ram_add  : top_col_ram_add) :
41                                      (s[0]?  top_col_ram_add :  top_col_ram_add));
42
43  assign type = s[2] ? (s[1] ? (s[0] ? top_type : top_type) :
44                               (s[0] ? rtb3_type : ltb3_type)) :
45                       (s[1] ? (s[0] ? h2_type  : v2_type) :
46                               (s[0]?  t1_type :  top_type));
47
48
49  //5 minor FSMs
50  v2term_fsm my_v2term_fsm(.clk(clk), .reset_sync(reset_sync),
51                           .start(v2_start), .finished(v2_finished),
52                           .row_ram_add(v2_row_ram_add),
53                           .type(v2_type), .row_ram_out(row_ram_out));
54
55
56  h2term_fsm my_h2term_fsm(.clk(clk), .reset_sync(reset_sync),
57                           .start(h2_start), .finished(h2_finished),
58                           .col_ram_add(h2_col_ram_add),
59                           .type(h2_type), .col_ram_out(col_ram_out));
60
61
62  ltb3_fsm my_ltb3_fsm(.clk(clk), .reset_sync(reset_sync),
```

```
63                          .start(ltb3_start), .finished(ltb3_finished),
64                          .row_ram_add(ltb3_row_ram_add), .col_ram_add(ltb3_col_ram_add),
65                          .type(ltb3_type),
66                          .row_ram_out(row_ram_out), .col_ram_out(col_ram_out));
67
68
69
70  rtb3_fsm my_rtb3_fsm(.clk(clk), .reset_sync(reset_sync),
71                          .start(rtb3_start), .finished(rtb3_finished),
72                          .row_ram_add(rtb3_row_ram_add), .col_ram_add(rtb3_col_ram_add),
73                          .type(rtb3_type),
74                          .row_ram_out(row_ram_out), .col_ram_out(col_ram_out));
75
76  t1term_fsm my_t1term_fsm(.clk(clk), .reset_sync(reset_sync),
77                          .start(t1_start), .finished(t1_finished),
78                          .row_ram_add(t1_row_ram_add),
79                          .type(t1_type), .row_ram_out(row_ram_out));
80
81
82
83  reg [5:0] state;
84  parameter idle                = {1'b0, 5'd0};
85  parameter delay0              = {1'b0, 5'd1};
86  parameter ld_edges_top_left0  = {1'b0, 5'd2};
87  parameter delay1              = {1'b0, 5'd26};
88  parameter ld_edges_top_left1  = {1'b0, 5'd27};
89  parameter delay2              = {1'b0, 5'd28};
90  parameter ld_edges_bot_right0 = {1'b0, 5'd29};
91  parameter delay3              = {1'b0, 5'd30};
92  parameter ld_edges_bot_right1 = {1'b0, 5'd31};
93  parameter analyze_edges       = {1'b0, 5'd4};
94  parameter n1                  = {1'b0, 5'd5};
95  parameter b1                  = {1'b0, 5'd6};
96  parameter t1                  = {1'b0, 5'd7};
97  parameter t2                  = {1'b0, 5'd8};
98  parameter tb1                 = {1'b0, 5'd9};
99  parameter tb2                 = {1'b0, 5'd10};
100 parameter r1                  = {1'b0, 5'd11};
101 parameter rb1                 = {1'b0, 5'd12};
102 parameter rt1                 = {1'b0, 5'd13};
103 parameter rtb1                = {1'b0, 5'd14};
104 parameter rtb2                = {1'b0, 5'd25};
105 parameter l1                  = {1'b0, 5'd15};
106 parameter lb1                 = {1'b0, 5'd16};
107 parameter lt1                 = {1'b0, 5'd17};
108 parameter ltb1                = {1'b0, 5'd18};
109 parameter ltb2                = {1'b0, 5'd19};
110 parameter lr1                 = {1'b0, 5'd20};
111 parameter lr2                 = {1'b0, 5'd21};
112 parameter lrb1                = {1'b0, 5'd22};
113 parameter lrt1                = {1'b0, 5'd23};
114 parameter lrtb1               = {1'b0, 5'd24};
115 parameter done                = {1'b1, 5'd25};
116
117 assign finished = state[5];
118
119 always @ (posedge clk) begin
120
121 case (state)
122 idle: begin
123
124 if (start) begin
125 s <= 0;
```

```
126   state <= delay0;
127   end
128   else begin
129   top_row_ram_add <= 6'd3;
130   top_col_ram_add <= 6'd3;
131   state <= idle;
132   end
133   end
134
135   delay0: state <= ld_edges_top_left0;
136
137   ld_edges_top_left0: begin //test top edge, left edge
138        edges[1] <= |row_ram_out[60:3];
139        edges[3] <= |col_ram_out[60:3];
140        top_row_ram_add <= 6'd4;
141        top_col_ram_add <= 6'd4;
142        state <= delay1;
143   end
144
145   delay1: state <= ld_edges_top_left1;
146
147   ld_edges_top_left1: begin //test top edge, left edge
148        edges[1] <= edges[1] && |row_ram_out[60:3];
149        edges[3] <= edges[3] && |col_ram_out[60:3];
150        top_row_ram_add <= 6'd59;
151        top_col_ram_add <= 6'd59;
152        state <= delay2;
153   end
154
155   delay2: state <= ld_edges_bot_right0;
156
157   ld_edges_bot_right0: begin //test bot edge, right edge
158        edges[0] <= |row_ram_out[60:35];
159        edges[2] <= |col_ram_out[60:27];
160        top_row_ram_add <= 6'd58;
161        top_col_ram_add <= 6'd58;
162        state <= delay3;
163   end
164
165   delay3: state <= ld_edges_bot_right1;
166
167   ld_edges_bot_right1: begin //test bot edge, right edge
168        edges[0] <= edges[0] && |row_ram_out[60:35];  //avoid text area
169        edges[2] <= edges[2] && |col_ram_out[60:27];
170        state <= analyze_edges;
171   end
172
173   analyze_edges:
174   case (edges)
175   4'b0000: state <= n1;
176   4'b0001: state <= b1;
177   4'b0010: state <= t1;
178   4'b0011: state <= tb1;
179   4'b0100: state <= r1;
180   4'b0101: state <= rb1;
181   4'b0110: state <= rt1;
182   4'b0111: state <= rtb1;
183   4'b1000: state <= l1;
184   4'b1001: state <= lb1;
185   4'b1010: state <= lt1;
186   4'b1011: state <= ltb1;
187   4'b1100: state <= lr1;
188   4'b1101: state <= lrb1;
```

```
189   4'b1110: state <= lrt1;
190   4'b1111: state <= lrtb1;
191   endcase
192
193   n1: begin
194   top_type <= 5'd0;//blank code
195   state <= done;
196   end
197
198   b1: begin
199   top_type <= 5'd1;//ps (bot_edge) code
200   state <= done;
201   end
202
203   t1: begin //t1 fsm
204   t1_start <= 1;
205   state <= t2;
206   s <= 1;
207   end
208
209   t2: begin
210   if (t1_finished) begin
211   state <= done;
212   end
213   else begin
214   state <= t2;
215   t1_start <= 0;
216   end
217   end
218
219
220   tb1: begin  //vert fsm
221   v2_start <= 1;
222   state <= tb2;
223   s <= 2;
224   end
225
226   tb2: begin
227   if (v2_finished) begin
228   state <= done;
229   end
230   else begin
231   state <= tb2;
232   v2_start <= 0;
233   end
234   end
235
236   r1: begin
237   top_type <= 5'd0;//none
238   state <= done;
239   end
240
241   rb1: begin
242   top_type <= 5'd23;//bottom right connector
243   state <= done;
244   end
245
246   rt1: begin
247   top_type <= 5'd24;//top right connector
248   state <= done;
249   end
250
251   rtb1: begin //rtb3 fsm
```

```
252   rtb3_start <= 1;
253   state <= rtb2;
254   s <= 5;
255   end
256
257   rtb2: begin
258   if (rtb3_finished) begin
259   state <= done;
260   end
261   else begin
262   state <= rtb2;
263   rtb3_start <= 0;
264   end
265   end
266
267   l1: begin
268   top_type <= 5'd0;//none
269   state <= done;
270   end
271
272   lb1: begin
273   top_type <= 5'd25;//lb connector
274   state <= done;
275   end
276
277   lt1: begin
278   top_type <= 5'd26;//lt connector
279   state <= done;
280   end
281
282   ltb1: begin  //ltb3 fsm
283   ltb3_start <= 1;
284   state <= ltb2;
285   s <= 4;
286   end
287
288   ltb2: begin
289   if (ltb3_finished) begin
290   state <= done;
291   end
292   else begin
293   state <= ltb2;
294   ltb3_start <= 0;
295   end
296   end
297
298
299   lr1: begin  //horizontal fsm
300   h2_start <= 1;
301   state <= lr2;
302   s <= 3;
303   end
304
305   lr2: begin
306   if (h2_finished) begin
307   state <= done;
308   end
309   else begin
310   h2_start <= 0;
311   state <= lr2;
312   end
313   end
314
```

```
315
316   lrb1: begin
317   top_type <= 5'd29;//lrb t connector
318   state <= done;
319   end
320
321   lrt1: begin
322   top_type <= 5'd30;//lrt t connector
323   state <= done;
324   end
325
326   lrtb1: begin
327   top_type <= 5'd31;//cross connector
328   state <= done;
329   end
330
331   done: begin
332   state <= idle;
333   end
334
335   endcase
336
337   if (reset_sync) begin
338   s <= 0;
339   state <= idle;
340   t1_start <= 0;
341   v2_start <= 0;
342   h2_start <= 0;
343   ltb3_start <= 0;
344   rtb3_start <= 0;
345   top_row_ram_add <= 6'd3;
346   top_col_ram_add <= 6'd3;
347   end
348
349   end
350   endmodule
351
352
```

```
1     module h2term_fsm(clk, reset_sync, start, finished,
2                       col_ram_add, type,
3                       col_ram_out);
4
5     input clk, start, reset_sync;
6     output finished;
7     reg finished;
8     output [5:0] col_ram_add;
9     reg [5:0] col_ram_add, row_ram_add;
10    input [63:0] col_ram_out;
11    output [4:0] type;
12    reg [4:0] type;
13
14    //used in calculating running sum, holds 60 inputs to summer
15    reg [59:0] gap_holder;
16    reg [9:0] ad0, ad1, ad2, ad3, ad4, ad5;
17    reg gapflag;
18    wire [5:0] sum;
19
20    wire [3:0] thickness;
21    reg [3:0] max_thickness;
22    wire [3:0] left, right;
23
24    //60 input 1bit adder
25    adder_10x6 my_adder(.row0(ad0), .row1(ad1), .row2(ad2), .row3(ad3),
26                        .row4(ad4), .row5(ad5), .sum(sum));
27
28    //find thickness of slice, here top most point minus bottom most point
29    row_thickness my_row_thickness(.row0(ad4), .row1(ad5), .left(left), .right(right), .thickn
      ess(thickness));
30
31
32    reg [4:0] state;
33    parameter idle = 0;
34    parameter cap_check0 = 1;
35    parameter delay0 = 2;
36    parameter sum0 = 3;
37    parameter sum1 = 4;
38    parameter sum2 = 5;
39    parameter find_start0 = 6;
40    parameter find_start1 = 7;
41    parameter gap_find0 = 8;
42    parameter done = 9;
43
44
45    wire [9:0] gap_row;
46    assign gap_row = {col_ram_out[row_ram_add-4],
47                      col_ram_out[row_ram_add-3],
48                      col_ram_out[row_ram_add-2],
49                      col_ram_out[row_ram_add-1],
50                      col_ram_out[row_ram_add],
51                      col_ram_out[row_ram_add+1],
52                      col_ram_out[row_ram_add+2],
53                      col_ram_out[row_ram_add+3],
54                      col_ram_out[row_ram_add+4],
55                      col_ram_out[row_ram_add+5]};
56
57
58    always @ (posedge clk) begin
59    case (state)
60    idle:
61         if (start) state <= cap_check0;
62         else begin
```

```
63              state <= idle;
64              col_ram_add <= 3;
65              row_ram_add <= 3;
66              gap_holder <= 0;
67              finished <= 0;
68              gapflag <= 0;
69              max_thickness <= 2;
70         end
71
72    cap_check0:
73         if (~|col_ram_out[60:3]) begin
74              //found capacitor
75              type <= 5'd7;
76              state <= done;
77         end
78         else if (&(col_ram_add[5:1])) begin
79              // not a cap, no discontinuity found
80              col_ram_add <= 4;
81              row_ram_add <= 4;
82              state <= delay0;
83         end
84         else begin
85              col_ram_add <= col_ram_add + 1;
86              state <= cap_check0;
87         end
88
89    delay0: state <= find_start0;
90
91    find_start0:
92         if (col_ram_out[row_ram_add]) begin
93         //found start of component
94              gap_holder <= gap_row;
95              col_ram_add <= col_ram_add + 1;
96              state <= find_start1;
97         end
98         else begin
99              row_ram_add <= row_ram_add + 1;
100             state <= find_start0;
101        end
102
103   find_start1: begin
104   gap_holder <= {gap_holder[49:0], gap_row};
105   col_ram_add <= col_ram_add + 1;
106   if (~|col_ram_add[2:0]) state <= sum0; //fill gap_holder before moving on
107   else state <= find_start1;
108   end
109
110   sum2: begin
111   gap_holder <= {gap_holder[49:0], gap_row}; //gap holder is shift register
112   state <= sum0;
113   end
114
115   sum0: begin
116   ad0 <= gap_holder[59:50]; //set inputs to adder
117   ad1 <= gap_holder[49:40];
118   ad2 <= gap_holder[39:30];
119   ad3 <= gap_holder[29:20];
120   ad4 <= gap_holder[19:10];
121   ad5 <= gap_holder[9:0];
122   state <= sum1;
123   end
124
125   sum1: begin
```

```
126   state <= gap_find0; //store max thickness
127   if (thickness > max_thickness && !(&thickness)) max_thickness <= thickness;
128   end
129
130   gap_find0:
131        if (&(col_ram_add[5:3])) begin
132             state <= done;
133             if (max_thickness < 4) type <= 5'd21; //wire, thin no gap
134             else if (gapflag) type <= 5'd9; //src, gap
135             else type <= 5'd5; //resistor, thick, no gap
136        end
137        else if ((sum < 12) && (thickness > 4)) begin
138             state <= sum2;
139             gapflag <= 1;
140             col_ram_add <= col_ram_add + 1;  //found gap
141        end
142        else if ((sum < 12) &&
143                 (thickness < 4) &&
144                 ( ((left > 7) && !(&left)) || (right < 2)) ) begin
145             state <= sum2;
146             gapflag <= 1;
147             col_ram_add <= col_ram_add + 1; //another way to find a gap...takes care of odd
    centering of
148                                             //wire on src, where there is a gap, but thickness
     is 1
149        end
150        else begin
151             state <= sum2;
152             col_ram_add <= col_ram_add + 1;
153        end
154
155   done: begin
156   state <= idle;
157   finished <= 1;
158   end
159
160   endcase
161   if (reset_sync) begin
162             state <= idle;
163             col_ram_add <= 3;
164             row_ram_add <= 3;
165             gap_holder <= 0;
166   ad0 <= 0;
167   ad1 <= 0;
168   ad2 <= 0;
169   ad3 <= 0;
170   ad4 <= 0;
171   ad5 <= 0;
172   finished <= 0;
173   gapflag <= 0;
174   max_thickness <= 2;
175   end
176
177   end
178   endmodule
179
```

```
1     module ltb3_fsm(clk, reset_sync, start, finished,
2                     row_ram_add, col_ram_add, type,
3                     row_ram_out, col_ram_out);
4
5     input clk, start, reset_sync;
6     output finished;
7     reg finished;
8     output [5:0] row_ram_add, col_ram_add;
9     reg [5:0] row_ram_add, col_ram_add, row_counter, col_counter;
10    input [63:0] row_ram_out, col_ram_out;
11    output [4:0] type;
12    reg [4:0] type;
13
14    reg [59:0] gap_holder;
15    reg [9:0] ad0, ad1, ad2, ad3, ad4, ad5;
16    wire [5:0] sum;
17
18    adder_10x6 my_adder(.row0(ad0), .row1(ad1), .row2(ad2), .row3(ad3),
19                        .row4(ad4), .row5(ad5), .sum(sum));
20
21    wire [9:0] gap_row;
22    assign gap_row = {col_ram_out[row_counter+4],
23                      col_ram_out[row_counter+3],
24                      col_ram_out[row_counter+2],
25                      col_ram_out[row_counter+1],
26                      col_ram_out[row_counter],
27                      col_ram_out[row_counter-1],
28                      col_ram_out[row_counter-2],
29                      col_ram_out[row_counter-3],
30                      col_ram_out[row_counter-4],
31                      col_ram_out[row_counter-5]};
32
33
34
35    reg [4:0] state;
36    parameter idle = 0;
37    parameter delay0 = 2;
38    parameter sum0 = 3;
39    parameter sum1 = 4;
40    parameter sum2 = 5;
41    parameter find_top_start0 = 6;
42    parameter find_left_start0 = 7;
43    parameter find_left_start1 = 10;
44    parameter gap_find0 = 8;
45    parameter done = 9;
46    parameter delay1 = 11;
47
48
49
50    always @ (posedge clk) begin
51    case (state)
52    idle:
53         if (start) state <= delay0;
54         else begin
55              state <= idle;
56              row_ram_add <= 4;
57              col_ram_add <= 4;
58              col_counter <= 59;
59              row_counter <= 59;
60              gap_holder <= 0;
61              ad0 <= 0;
62              ad1 <= 0;
63              ad2 <= 0;
```

```
64              ad3 <= 0;
65              ad4 <= 0;
66              ad5 <= 0;
67              finished <= 0;
68         end
69
70    delay0: state <= find_top_start0;
71
72    find_top_start0:
73         if (row_ram_out[col_counter]) begin
74         col_counter <= ~col_counter;
75         state <= find_left_start0;
76         end
77         else begin
78              col_counter <= col_counter - 1;
79              state <= find_top_start0;
80         end
81
82    find_left_start0:
83         if (col_ram_out[row_counter]) state <= find_left_start1;
84         else begin
85              row_counter <= row_counter - 1;
86              state <= find_left_start0;
87         end
88
89    delay1: state <= find_left_start1;
90
91    find_left_start1: begin    //fill gap_holder
92         gap_holder <= {gap_holder[49:0], gap_row};
93         col_ram_add <= col_ram_add + 1;
94         if (col_ram_add == 9) state <= sum0;
95         else state <= delay1;
96    end
97
98    sum2: begin
99    gap_holder <= {gap_holder[49:0], gap_row};
100   state <= sum0;
101   end
102
103   sum0: begin
104   ad0 <= gap_holder[59:50];
105   ad1 <= gap_holder[49:40];
106   ad2 <= gap_holder[39:30];
107   ad3 <= gap_holder[29:20];
108   ad4 <= gap_holder[19:10];
109   ad5 <= gap_holder[9:0];
110   state <= sum1;
111   end
112
113   sum1: state <= gap_find0;
114
115   gap_find0:
116        if (col_ram_add == col_counter) begin
117             state <= done; //we have reached the rightmost point of the component without
118             type <= 5'd28; //finding our large line indicating transistor...wire
119        end
120        else if (sum > 16) begin
121             state <= done;
122             type <= 5'd10; //found large line at base of transistor...transistor
123        end
124        else begin
125             state <= sum2;
126             col_ram_add <= col_ram_add + 1;
```

```
127         end
128
129
130     done: begin
131     state <= idle;
132     finished <= 1;
133     end
134
135     endcase
136     if (reset_sync) begin
137               state <= idle;
138               row_ram_add <= 4;
139               col_ram_add <= 4;
140               col_counter <= 59;
141               row_counter <= 59;
142               gap_holder <= 0;
143               finished <= 0;
144               ad0 <= 0;
145               ad1 <= 0;
146               ad2 <= 0;
147               ad3 <= 0;
148               ad4 <= 0;
149               ad5 <= 0;
150     end
151
152
153     end
154     endmodule
155
156
```

```
1   module mem_fsm(clk, reset_sync, start, finished,
2                  row_num, col_num,
3                  rom_add,
4                  row_ram_add, row_ram_out, row_ram_we,
5                  col_ram_add, col_ram_in, col_ram_we);
6
7   input clk, reset_sync, start;
8   output finished;
9
10  input [2:0] row_num, col_num;
11  output [11:0] rom_add;
12  output [5:0] row_ram_add, col_ram_add;
13  reg [5:0] row_ram_add, col_ram_add;
14  assign rom_add = 512*row_num + col_num + 8*row_ram_add;  //addressing logic for rowram
15
16  input [63:0] row_ram_out;
17  output [63:0] col_ram_in;
18  reg [63:0] col_ram_in;
19
20  output row_ram_we, col_ram_we;
21
22  reg [5:0] state;
23
24  parameter idle           = {1'b0, 1'b0, 1'b0, 3'd0};
25  parameter delay0         = {1'b0, 1'b0, 1'b0, 3'd1};
26  parameter write_row_ram  = {1'b0, 1'b1, 1'b0, 3'd0};
27  parameter delay1         = {1'b0, 1'b0, 1'b0, 3'd2};
28  parameter load_reg       = {1'b0, 1'b0, 1'b0, 3'd3};
29  parameter write_col_ram  = {1'b0, 1'b0, 1'b1, 3'd0};
30  parameter done           = {1'b1, 1'b0, 1'b0, 3'd0};
31
32  assign finished = state [5];
33  assign row_ram_we = state[4];
34  assign col_ram_we = state[3];
35
36  always @ (posedge clk) begin
37
38  case (state)
39
40  idle: begin
41       if (start) begin
42            state <= delay0;
43       end
44       else begin
45            row_ram_add <= 0;
46            col_ram_add <= 0;
47            state <= idle;
48            col_ram_in <= 0;
49       end
50  end
51
52  delay0: state <= write_row_ram;
53
54  write_row_ram: begin
55       if (&(row_ram_add)) begin
56            row_ram_add <= row_ram_add + 1;
57            state <= delay1;
58       end
59       else begin
60            row_ram_add <= row_ram_add + 1;
61            state <= delay0;
62       end
63  end
```

```
64
65    delay1: state <= load_reg;
66
67    load_reg: begin  //fill shift register, creating columns
68         if (&(row_ram_add)) begin
69              state <= write_col_ram;
70              row_ram_add <= row_ram_add + 1;
71              col_ram_in <= {col_ram_in[62:0], row_ram_out[~col_ram_add]};
72         end
73         else begin
74              col_ram_in <= {col_ram_in[62:0], row_ram_out[~col_ram_add]};
75              row_ram_add <= row_ram_add + 1;
76              state <= delay1;
77         end
78    end
79
80    write_col_ram: begin
81         if (&(col_ram_add)) begin
82              state <= done;
83         end
84         else begin
85              col_ram_add <= col_ram_add + 1;
86              state <= delay1;
87         end
88    end
89
90    done: begin
91         state <= idle;
92    end
93
94    endcase
95
96    if (reset_sync) begin
97         state <= idle;
98         row_ram_add <= 0;
99         col_ram_add <= 0;
100        col_ram_in <= 0;
101        end
102
103   end
104   endmodule
105
106
```

```
1   module ravi_fsm(clk,reset_sync,finished,
2                   rom_add,
3                   row_ram_add, row_ram_out, row_ram_we,
4                   col_ram_add, col_ram_in, col_ram_out, col_ram_we,
5                   results_ram_add, results_ram_in, results_ram_we,
6                   //debug
7                    val_reg, val_pads, mult_pads
8                   //end debug
9                   );
10
11      input clk, reset_sync;
12      output finished; //finished signal indicates Vijay's stuff can begin
13      output [11:0] rom_add;
14      output [5:0] row_ram_add, col_ram_add, results_ram_add;
15      input [63:0] row_ram_out,col_ram_out;
16      output [63:0] col_ram_in;
17      output [19:0] results_ram_in;
18      output row_ram_we, col_ram_we, results_ram_we;
19
20      wire mem_handle_start, comp_recog_start, text_recog_start;
21      wire mem_handle_finished, comp_recog_finished, text_recog_finished;
22
23      wire mem_handle_row_ram_we, mem_handle_col_ram_we;
24      reg results_ram_we;
25
26
27      reg [5:0] blockcount; //blockcount = gridblock number
28      wire [2:0] row_num, col_num;
29      assign col_num = blockcount[2:0]; //numbering across starting from 0
30      assign row_num = blockcount[5:3];
31
32      wire [5:0] mem_handle_row_ram_add, mem_handle_col_ram_add;  //allow minor FSMs to
33      wire [5:0] comp_recog_row_ram_add, comp_recog_col_ram_add;  //speak to RAMs
34      wire [5:0] text_recog_row_ram_add, text_recog_col_ram_add;
35
36      wire [4:0] type;
37      wire [11:0] val;
38      wire [2:0] mult;
39
40      assign results_ram_in = {type, val, mult}; //format of result
41
42      assign results_ram_add = blockcount; //addressed by grid block number
43
44
45
46      mem_fsm my_mem_fsm(.clk(clk), .reset_sync(reset_sync),
47                         .start(mem_handle_start), .finished(mem_handle_finished),
48                         .row_num(row_num), .col_num(col_num),
49                         .rom_add(rom_add),
50                         .row_ram_add(mem_handle_row_ram_add),
51                         .row_ram_out(row_ram_out),
52                         .row_ram_we(mem_handle_row_ram_we),
53                         .col_ram_add(mem_handle_col_ram_add),
54                         .col_ram_in(col_ram_in),
55                         .col_ram_we(mem_handle_col_ram_we));
56
57
58
59
60      choose_fsm my_choose_fsm(.clk(clk), .reset_sync(reset_sync),
61                               .start(comp_recog_start), .finished(comp_recog_finished),
62                               .row_ram_add(comp_recog_row_ram_add),
63                               .col_ram_add(comp_recog_col_ram_add),
```

```
64                              .type(type),
65                              .row_ram_out(row_ram_out), .col_ram_out(col_ram_out));
66
67
68
69      output [79:0] val_reg;
70      output [9:0] val_pads;
71      output [6:0] mult_pads;
72
73      txt_fsm my_txt_fsm(.clk(clk), .reset_sync(reset_sync),
74                         .start(text_recog_start), .finished(text_recog_finished),
75                         .row_ram_add(text_recog_row_ram_add), .row_ram_out(row_ram_out),
76                         .col_ram_add(text_recog_col_ram_add),
77                         .col_ram_out(col_ram_out),
78                         .val(val), .mult(mult), .val_reg(val_reg), .val_pads(val_pads), .mu
   lt_pads(mult_pads));
79
80
81       reg [9:0] state;
82      //{extra distinction, finished, what state?, we?, numbering}
83
84      parameter idle            = {1'b0, 1'b0, 3'b000, 3'b000, 2'b00};
85      parameter mem_handle0     = {1'b0, 1'b0, 3'b001, 3'b001, 2'b00};
86      parameter mem_handle1     = {1'b0, 1'b0, 3'b001, 3'b000, 2'b00};
87      parameter comp_recog0     = {1'b0, 1'b0, 3'b010, 3'b010, 2'b00};
88      parameter comp_recog1     = {1'b0, 1'b0, 3'b010, 3'b000, 2'b00};
89      parameter text_recog0     = {1'b0, 1'b0, 3'b100, 3'b100, 2'b00};
90      parameter text_recog1     = {1'b0, 1'b0, 3'b100, 3'b000, 2'b00};
91      parameter results0        = {1'b0, 1'b0, 3'b000, 3'b000, 2'b01};
92      parameter results1        = {1'b0, 1'b0, 3'b000, 3'b000, 2'b10};
93      parameter results2        = {1'b0, 1'b0, 3'b000, 3'b000, 2'b11};
94      parameter scan1           = {1'b1, 1'b0, 3'b000, 3'b000, 2'b00};
95      parameter scan2           = {1'b1, 1'b0, 3'b000, 3'b000, 2'b01};
96      parameter scan3           = {1'b1, 1'b0, 3'b000, 3'b000, 2'b10};
97      parameter done            = {1'b0, 1'b1, 3'b000, 3'b000, 2'b00};
98
99
100     //handle control of RAMs
101
102     assign row_ram_add = state[7] ? text_recog_row_ram_add :
103                          (state[6] ? comp_recog_row_ram_add :
104                          (state[5] ? mem_handle_row_ram_add : 6'b0));
105
106      assign col_ram_add = state[7] ? text_recog_col_ram_add :
107                          (state[6] ? comp_recog_col_ram_add :
108                          (state[5] ? mem_handle_col_ram_add : 6'b0));
109
110
111
112     assign row_ram_we = state[5] ? mem_handle_row_ram_we : 1'b0;
113     assign col_ram_we = state[5] ? mem_handle_col_ram_we : 1'b0;
114     assign finished = state[8];
115     assign text_recog_start = state[4];
116     assign comp_recog_start = state[3];
117     assign mem_handle_start = state[2];
118
119     always @ (posedge clk) begin
120     case (state)
121
122     idle: state <= mem_handle0;
123
124     mem_handle0: state <= mem_handle1;  //start mem_fsm
125     mem_handle1: if (mem_handle_finished) state <= comp_recog0;
```

```
126     else state <= mem_handle1;
127
128     comp_recog0: state <= comp_recog1; //start choose_fsm
129     comp_recog1: if (comp_recog_finished) state <= text_recog0;
130     else state <= comp_recog1;
131
132     text_recog0: state <= text_recog1;  //start txt_fsm
133     text_recog1: if (text_recog_finished) state <= results0;
134     else state <= text_recog1;
135
136     results0: begin  //start writing results
137     results_ram_we <= 1;
138     state <= results1;
139     end
140
141     results1: begin
142     results_ram_we <= 0;
143     state <= results2;
144     end
145
146     results2: //next grid block or done?
147     if (&(blockcount)) begin
148     state <= scan1;
149     blockcount <= blockcount + 1;
150     end
151     else begin
152     blockcount <= blockcount + 1;
153     state <= idle;
154     end
155
156     scan1: state <= scan2; //quick debug states
157     scan2: state <= scan3;
158     scan3: if (&(blockcount)) state <= done;
159     else begin
160     blockcount <= blockcount + 1;
161     state <= scan1;
162     end
163
164
165
166     done: state <= done;
167
168     endcase
169     if (reset_sync) begin
170     state <= idle;
171     blockcount <= 0;
172     results_ram_we <= 0;
173     end
174
175     end
176     endmodule
177
```

```
1   module row_thickness(row0, row1, left, right, thickness);
2       input [9:0] row0, row1;
3       //find left most point minus right most point
4       output [3:0] thickness;
5
6       output [3:0] left, right;
7       assign left = (row0[9] || row1[9]) ? 4'd9 :
8                     ((row0[8] || row1[8]) ? 4'd8 :
9                          ((row0[7] || row1[7]) ? 4'd7 :
10                              ((row0[6] || row1[6]) ? 4'd6 :
11                                  ((row0[5] || row1[5]) ? 4'd5 :
12                                      ((row0[4] || row1[4]) ? 4'd4 :
13                                          ((row0[3] || row1[3]) ? 4'd3 :
14                                              ((row0[2] || row1[2]) ? 4'd2 :
15                                                  ((row0[1] || row1[1]) ? 4'd1 :
16                                                      ((row0[0] || row1[0]) ? 4'd0 :
     4'd15))
17                                                  )
18                                          )
19                                      )
20                                  )
21                              )
22                          )
23                    );
24
25      assign right = (row0[0] || row1[0]) ? 4'd0 :
26                    ((row0[1] || row1[1]) ? 4'd1 :
27                         ((row0[2] || row1[2]) ? 4'd2 :
28                             ((row0[3] || row1[3]) ? 4'd3 :
29                                 ((row0[4] || row1[4]) ? 4'd4 :
30                                     ((row0[5] || row1[5]) ? 4'd5 :
31                                         ((row0[6] || row1[6]) ? 4'd6 :
32                                             ((row0[7] || row1[7]) ? 4'd7 :
33                                                 ((row0[8] || row1[8]) ? 4'd8 :
34                                                     ((row0[9] || row1[9]) ? 4'd9 :
     4'd0))
35                                                 )
36                                         )
37                                     )
38                                 )
39                             )
40                         )
41                    );
42
43
44
45
46
47
48
49
50
51      assign thickness = left-right;
52
53  endmodule
54
```

```
1     module rtb3_fsm(clk, reset_sync, start, finished,
2                     row_ram_add, col_ram_add, type,
3                     row_ram_out, col_ram_out);
4
5     input clk, start, reset_sync;
6     output finished;
7     reg finished;
8     output [5:0] row_ram_add, col_ram_add;
9     reg [5:0] row_ram_add, col_ram_add, row_counter, col_counter;
10    input [63:0] row_ram_out, col_ram_out;
11    output [4:0] type;
12    reg [4:0] type;
13
14    reg [59:0] gap_holder;
15    reg [9:0] ad0, ad1, ad2, ad3, ad4, ad5;
16    wire [5:0] sum;
17
18    adder_10x6 my_adder(.row0(ad0), .row1(ad1), .row2(ad2), .row3(ad3),
19                        .row4(ad4), .row5(ad5), .sum(sum));
20
21    wire [9:0] gap_row;
22    assign gap_row = {col_ram_out[row_counter+4],
23                      col_ram_out[row_counter+3],
24                      col_ram_out[row_counter+2],
25                      col_ram_out[row_counter+1],
26                      col_ram_out[row_counter],
27                      col_ram_out[row_counter-1],
28                      col_ram_out[row_counter-2],
29                      col_ram_out[row_counter-3],
30                      col_ram_out[row_counter-4],
31                      col_ram_out[row_counter-5]};
32
33
34
35    reg [4:0] state;
36    parameter idle = 0;
37    parameter delay0 = 2;
38    parameter sum0 = 3;
39    parameter sum1 = 4;
40    parameter sum2 = 5;
41    parameter find_top_start0 = 6;
42    parameter find_right_start0 = 7;
43    parameter find_right_start1 = 10;
44    parameter gap_find0 = 8;
45    parameter done = 9;
46    parameter delay1 = 11;
47
48
49
50    always @ (posedge clk) begin
51    case (state)
52    idle:
53         if (start) state <= delay0;
54         else begin
55              state <= idle;
56              row_ram_add <= 4;
57              col_ram_add <= 59;
58              col_counter <= 59;
59              row_counter <= 59;
60              gap_holder <= 0;
61              ad0 <= 0;
62              ad1 <= 0;
63              ad2 <= 0;
```

```
64              ad3 <= 0;
65              ad4 <= 0;
66              ad5 <= 0;
67              finished <= 0;
68         end
69
70    delay0: state <= find_top_start0;
71
72    find_top_start0:
73         if (row_ram_out[col_counter]) begin
74         col_counter <= ~col_counter;
75         state <= find_right_start0;
76         end
77         else begin
78              col_counter <= col_counter - 1;
79              state <= find_top_start0;
80         end
81
82    find_right_start0:
83         if (col_ram_out[row_counter]) state <= find_right_start1;
84         else begin
85              row_counter <= row_counter - 1;
86              state <= find_right_start0;
87         end
88
89    delay1: state <= find_right_start1;
90
91    find_right_start1: begin
92         gap_holder <= {gap_row, gap_holder[59:10]};
93         col_ram_add <= col_ram_add - 1;
94         if (col_ram_add == 54) state <= sum0;
95         else state <= delay1;
96    end
97
98    sum2: begin
99    gap_holder <= {gap_row, gap_holder[59:10]};
100   state <= sum0;
101   end
102
103   sum0: begin
104   ad0 <= gap_holder[59:50];
105   ad1 <= gap_holder[49:40];
106   ad2 <= gap_holder[39:30];
107   ad3 <= gap_holder[29:20];
108   ad4 <= gap_holder[19:10];
109   ad5 <= gap_holder[9:0];
110   state <= sum1;
111   end
112
113   sum1: state <= gap_find0;
114
115   gap_find0:
116        if (col_ram_add == col_counter) begin
117             state <= done;      //we have reached the leftmost point of the component withou
      t
118             type <= 5'd27;      //finding our large line indicating transistor...wire
119        end
120        else if (sum > 16) begin
121             state <= done;     //found large line at base of transistor...transistor
122             type <= 5'd11;
123        end
124        else begin
125             state <= sum2;
```

```
126             col_ram_add <= col_ram_add - 1;
127         end
128
129
130     done: begin
131     state <= idle;
132     finished <= 1;
133     end
134
135     endcase
136     if (reset_sync) begin
137             state <= idle;
138             row_ram_add <= 4;
139             col_ram_add <= 59;
140             col_counter <= 59;
141             row_counter <= 59;
142             gap_holder <= 0;
143             finished <= 0;
144             ad0 <= 0;
145             ad1 <= 0;
146             ad2 <= 0;
147             ad3 <= 0;
148             ad4 <= 0;
149             ad5 <= 0;
150     end
151
152
153     end
154     endmodule
155
156
```

```
1   module t1term_fsm(clk, reset_sync, start, finished,
2                     row_ram_add, type,
3                     row_ram_out);
4
5   input clk, start, reset_sync;
6   output finished;
7   reg finished;
8   output [5:0] row_ram_add;
9   reg [5:0] row_ram_add;
10  input [63:0] row_ram_out;
11
12  output [4:0] type;
13  reg [4:0] type;
14
15  reg [1:0] state;
16
17
18  parameter idle = 0;
19  parameter gnd_check0 = 1;
20  parameter gnd_check1 = 2;
21  parameter done = 3;
22
23  always @ (posedge clk) begin
24  case (state)
25  idle: if (start) state <= gnd_check0;
26  else begin
27  state <= idle;
28  row_ram_add <= 1;
29  finished <= 0;
30  end
31
32
33  gnd_check0: begin
34  if (~|row_ram_out[60:35]) begin
35  //found a discontinuity
36  state <= gnd_check1;
37  end
38  else begin
39  //haven't finished scanning rows
40  row_ram_add <= row_ram_add + 1;
41  end
42  end
43
44  gnd_check1: begin
45  if (|row_ram_out[60:35]) begin
46  //found second part of gnd node
47  type <= 5'd3;//gnd node
48  state <= done;
49  end
50  else if (&(row_ram_add[5:1])) begin
51  //finished scanning rows
52  type <= 5'd2;//power supply
53  state <= done;
54  end
55  else begin
56  //haven't finished scanning rows
57  row_ram_add <= row_ram_add + 1;
58  state <= gnd_check1;
59  end
60  end
61
62  done: begin
63  state <= idle;
```

```
64      finished <= 1;
65      end
66
67      endcase
68
69      if (reset_sync) begin
70      row_ram_add <= 1;
71      state <= idle;
72      finished <= 0;
73      end
74
75      end
76      endmodule
77
```

```
1     module v2term_fsm(clk, reset_sync, start, finished,
2                       row_ram_add, type,
3                       row_ram_out);
4
5     input clk, start, reset_sync;
6     output finished;
7     reg finished;
8     output [5:0] row_ram_add;
9     reg [5:0] row_ram_add, col_ram_add;
10    input [63:0] row_ram_out;
11    output [4:0] type;
12    reg [4:0] type;
13
14    reg [59:0] gap_holder;
15    reg [9:0] ad0, ad1, ad2, ad3, ad4, ad5;
16    reg gapflag;
17    wire [5:0] sum;
18
19    wire [3:0] thickness;
20    reg [3:0] max_thickness;
21    wire [3:0] left, right;
22
23
24    adder_10x6 my_adder(.row0(ad0), .row1(ad1), .row2(ad2), .row3(ad3),
25                        .row4(ad4), .row5(ad5), .sum(sum));
26
27    row_thickness my_row_thickness(.row0(ad4), .row1(ad5), .left(left), .right(right), .thickn
      ess(thickness));
28
29
30    reg [4:0] state;
31    parameter idle = 0;
32    parameter cap_check0 = 1;
33    parameter delay0 = 2;
34    parameter sum0 = 3;
35    parameter sum1 = 4;
36    parameter sum2 = 5;
37    parameter find_start0 = 6;
38    parameter find_start1 = 7;
39    parameter gap_find0 = 8;
40    parameter done = 9;
41
42
43
44
45    wire [9:0] gap_row;
46    assign gap_row = {row_ram_out[col_ram_add-4],
47                      row_ram_out[col_ram_add-3],
48                      row_ram_out[col_ram_add-2],
49                      row_ram_out[col_ram_add-1],
50                      row_ram_out[col_ram_add],
51                      row_ram_out[col_ram_add+1],
52                      row_ram_out[col_ram_add+2],
53                      row_ram_out[col_ram_add+3],
54                      row_ram_out[col_ram_add+4],
55                      row_ram_out[col_ram_add+5]};
56
57
58    always @ (posedge clk) begin
59    case (state)
60    idle:
61         if (start) state <= cap_check0;
62         else begin
```

```
63              state <= idle;
64              row_ram_add <= 3;
65              col_ram_add <= 3;
66              gap_holder <= 0;
67              finished <= 0;
68              gapflag <= 0;
69              max_thickness <= 2;
70         end
71
72    cap_check0:
73         if (~|row_ram_out[60:3]) begin
74              //found capacitor
75              type <= 5'd6;
76              state <= done;
77         end
78         else if (&(row_ram_add[5:1])) begin
79              // not a cap, no discontinuity found
80              row_ram_add <= 4;
81              col_ram_add <= 4;
82              state <= delay0;
83         end
84         else begin
85              row_ram_add <= row_ram_add + 1;
86              state <= cap_check0;
87         end
88
89    delay0: state <= find_start0;
90
91    find_start0:
92         if (row_ram_out[col_ram_add]) begin
93         //found start
94              gap_holder <= gap_row;
95              row_ram_add <= row_ram_add + 1;
96              state <= find_start1;
97         end
98         else begin
99              col_ram_add <= col_ram_add + 1;
100             state <= find_start0;
101        end
102
103   find_start1: begin
104   gap_holder <= {gap_holder[49:0], gap_row};
105   row_ram_add <= row_ram_add + 1;
106   if (~|row_ram_add[2:0]) state <= sum0;
107   else state <= find_start1;
108   end
109
110   sum2: begin
111   gap_holder <= {gap_holder[49:0], gap_row};
112   state <= sum0;
113   end
114
115   sum0: begin
116   ad0 <= gap_holder[59:50];
117   ad1 <= gap_holder[49:40];
118   ad2 <= gap_holder[39:30];
119   ad3 <= gap_holder[29:20];
120   ad4 <= gap_holder[19:10];
121   ad5 <= gap_holder[9:0];
122   state <= sum1;
123   end
124
125   sum1: begin
```

```
126   state <= gap_find0;
127   if (thickness > max_thickness && !(&thickness)) max_thickness <= thickness;
128   end
129
130
131   gap_find0:
132        if (&(row_ram_add[5:3])) begin
133             state <= done;
134             if (max_thickness < 4) type <= 5'd22;
135             else if (gapflag) type <= 5'd8;
136             else type <= 5'd4;
137        end
138        else if ((sum < 12) && (thickness > 4)) begin
139             state <= sum2;
140             gapflag <= 1;
141             row_ram_add <= row_ram_add + 1;
142        end
143        else if ((sum < 12) &&
144                 ( ((left > 8) && !(&left)) ||
145                   ((right < 1) && (&right))) ) begin
146             state <= sum2;
147             gapflag <= 1;
148             col_ram_add <= col_ram_add + 1;
149        end
150
151        else begin
152             state <= sum2;
153             row_ram_add <= row_ram_add + 1;
154        end
155
156   done: begin
157   state <= idle;
158   finished <= 1;
159   end
160
161   endcase
162   if (reset_sync) begin
163             state <= idle;
164             row_ram_add <= 1;
165             col_ram_add <= 1;
166             gap_holder <= 0;
167   ad0 <= 0;
168   ad1 <= 0;
169   ad2 <= 0;
170   ad3 <= 0;
171   ad4 <= 0;
172   ad5 <= 0;
173   finished <= 0;
174   gapflag <= 0;
175   max_thickness <= 2;
176   end
177
178   end
179   endmodule
180
```

```
////////////////////////////////////////////////////////////////////////////////
// Copyright (c) 1995-2003 Xilinx, Inc.
// All Right Reserved.
////////////////////////////////////////////////////////////////////////////////
//   ____  ____
//  /   /\/   /
// /___/  \  /    Vendor: Xilinx
// \   \   \/     Version : 6.3.03i
//  \   \         Application :
//  /   /         Filename : pixel_clock.v
// /___/   /\     Timestamp : 05/09/2005 16:59:57
// \   \  /  \
//  \___\/\___\
//
//Command:
//Design Name: pixel_clock
//
// Module pixel_clock
// Generated by Xilinx Architecture Wizard
// Written for synthesis tool: XST
`timescale 1ns / 1ps

module pixel_clock(CLKIN_IN,
                   RST_IN,
                   CLKFX_OUT,
                   LOCKED_OUT);

    input CLKIN_IN;
    input RST_IN;
   output CLKFX_OUT;
   output LOCKED_OUT;

   wire CLKFX_BUF;
   wire GND;

   assign GND = 0;
   BUFG CLKFX_BUFG_INST (.I(CLKFX_BUF),
                         .O(CLKFX_OUT));
   // Period Jitter (unit interval) for block DCM_INST = 0.05 UI
   // Period Jitter (Peak-to-Peak) for block DCM_INST = 1.06 ns
   DCM DCM_INST (.CLKFB(GND),
                 .CLKIN(CLKIN_IN),
                 .DSSEN(GND),
                 .PSCLK(GND),
                 .PSEN(GND),
                 .PSINCDEC(GND),
                 .RST(RST_IN),
                 .CLKDV(),
                 .CLKFX(CLKFX_BUF),
                 .CLKFX180(),
                 .CLK0(),
                 .CLK2X(),
                 .CLK2X180(),
                 .CLK90(),
                 .CLK180(),
                 .CLK270(),
                 .LOCKED(LOCKED_OUT),
                 .PSDONE(),
                 .STATUS());
   // synthesis attribute CLK_FEEDBACK of DCM_INST is "NONE"
   // synthesis attribute CLKDV_DIVIDE of DCM_INST is "2.000000"
   // synthesis attribute CLKFX_DIVIDE of DCM_INST is "7"
   // synthesis attribute CLKFX_MULTIPLY of DCM_INST is "13"
   // synthesis attribute CLKIN_DIVIDE_BY_2 of DCM_INST is "FALSE"
   // synthesis attribute CLKIN_PERIOD of DCM_INST is "37.037000"
   // synthesis attribute CLKOUT_PHASE_SHIFT of DCM_INST is "NONE"
   // synthesis attribute DESKEW_ADJUST of DCM_INST is "SYSTEM_SYNCHRONOUS"
   // synthesis attribute DFS_FREQUENCY_MODE of DCM_INST is "LOW"
   // synthesis attribute DLL_FREQUENCY_MODE of DCM_INST is "LOW"
   // synthesis attribute DUTY_CYCLE_CORRECTION of DCM_INST is "TRUE"
   // synthesis attribute FACTORY_JF of DCM_INST is "C080"
   // synthesis attribute PHASE_SHIFT of DCM_INST is "0"
   // synthesis attribute STARTUP_WAIT of DCM_INST is "FALSE"
   // synopsys translate_off
   defparam DCM_INST.CLK_FEEDBACK = "NONE";
   defparam DCM_INST.CLKDV_DIVIDE = 2.000000;
   defparam DCM_INST.CLKFX_DIVIDE = 7;
   defparam DCM_INST.CLKFX_MULTIPLY = 13;
   defparam DCM_INST.CLKIN_DIVIDE_BY_2 = "FALSE";
   defparam DCM_INST.CLKIN_PERIOD = 37.037000;
   defparam DCM_INST.CLKOUT_PHASE_SHIFT = "NONE";
   defparam DCM_INST.DESKEW_ADJUST = "SYSTEM_SYNCHRONOUS";
   defparam DCM_INST.DFS_FREQUENCY_MODE = "LOW";
   defparam DCM_INST.DLL_FREQUENCY_MODE = "LOW";
   defparam DCM_INST.DUTY_CYCLE_CORRECTION = "TRUE";
   defparam DCM_INST.FACTORY_JF = 16'hC080;
   defparam DCM_INST.PHASE_SHIFT = 0;
   defparam DCM_INST.STARTUP_WAIT = "FALSE";
   // synopsys translate_on
endmodule
```

```
module video_top(clk, reset, switches, ideal_data, raw_ckt_dout, serial_export, vga_out_hsync, vga_out_vsync, vga_out_blank_b, vga_out_sync_b, vga_out_r
ed, vga_out_green, vga_out_blue, ideal_addr, raw_ckt_addr, rs232_rts, rs232_txd);
   input clk, reset;
   input [1:0] switches;
   input [19:0] ideal_data;
   input [63:0] raw_ckt_dout;
   input        serial_export;

   output vga_out_hsync, vga_out_vsync, vga_out_blank_b, vga_out_sync_b;
   output [7:0] vga_out_red, vga_out_green, vga_out_blue;
   output [5:0] ideal_addr;
   output [11:0] raw_ckt_addr;
   output        rs232_rts, rs232_txd;


   // synchronizer
   /*
   wire          reset_sync;
   wire [1:0]    control_sync;
   synchronizer s(.clk(clk),
                  .reset(reset),
                  .control(switches),
                  .reset_sync(reset_sync),
                  .control_sync(control_sync));
                  */

   // component rom
   wire [13:0] comp_rom_addr;
   wire [7:0]  comp_rom_data;

   // instantiate component rom
   component_rom comp_rom(.clk(clk),
                          .addr(comp_rom_addr),
                          .dout(comp_rom_data));

   // character rom
   wire [7:0]  char_rom_addr;
   wire [7:0]  char_rom_dout;

   // instantiate character rom
   character_rom char_rom(.clk(clk),
                          .addr(char_rom_addr),
                          .dout(char_rom_dout));

   // node value ram
   wire [6:0]  node_addr, node_data_write, node_data_read;
   wire        node_data_we;

   node_value nv(.addr(node_addr),
                 .clk(clk),
                 .din(node_data_write),
                 .dout(node_data_read),
                 .we(node_data_we));

   // spice text ram
   wire [10:0] spice_addr;
   wire [4:0]  spice_data_write, spice_data_read;
   wire        spice_we;

   spice_ram_5x1344 spiceram(.addr(spice_addr),
                             .clk(clk),
                             .din(spice_data_write),
                             .dout(spice_data_read),
                             .we(spice_we));


   // video ram
   wire [15:0] vram_write_addr, vram_read_addr;
   wire [7:0]  vram_write_data, vram_read_data;
   wire        vram_write_we;

   // instantiate video ram
   videoram vr(.addra(vram_write_addr),
               .addrb(vram_read_addr),
               .clka(clk),
               .clkb(clk),
               .dina(vram_write_data),
               .doutb(vram_read_data),
               .wea(vram_write_we));


   // vram display
   vram_display vd(.clk(clk),
                   .reset(reset),
                   .vram_read_data(vram_read_data),
                   .vga_out_hsync(vga_out_hsync),
                   .vga_out_vsync(vga_out_vsync),
                   .vga_out_blank_b(vga_out_blank_b),
                   .vga_out_sync_b(vga_out_sync_b),
                   .vga_out_red(vga_out_red),
                   .vga_out_green(vga_out_green),
                   .vga_out_blue(vga_out_blue),
                   .vram_read_addr(vram_read_addr));

   // major fsm
   major_fsm mf(.clk(clk),
                .reset(reset),
                .control(switches),
                .raw_ckt_dout(raw_ckt_dout),
                .ideal_data(ideal_data),
                .comp_rom_data(comp_rom_data),
                .char_rom_dout(char_rom_dout),
                .node_data_read(node_data_read),
                .spice_data_read(spice_data_read),
                .serial_export(serial_export),
                .raw_ckt_addr(raw_ckt_addr),
                .ideal_addr(ideal_addr),
                .comp_rom_addr(comp_rom_addr),
                .char_rom_addr(char_rom_addr),
                .node_data_write(node_data_write),
                .node_data_we(node_data_we),
                .node_addr(node_addr),
                .spice_data_write(spice_data_write),
                .spice_addr(spice_addr),
                .spice_we(spice_we),
                .vram_write_we(vram_write_we),
                .vram_write_data(vram_write_data),
                .vram_write_addr(vram_write_addr),
                .rs232_txd(rs232_txd),
                .rs232_rts(rs232_rts)
                );
endmodule // video_top
```

```
// debug
module sync_generator(clk, reset, hsyncbar, vsyncbar, blankbar, cnt, vcnt, state);
   // end debug
   // module sync_generator(clk, reset, hsyncbar, vsyncbar);
   input clk, reset;
   output hsyncbar, vsyncbar, blankbar;
   // debug
   output [9:0] cnt;
   output [9:0] vcnt;
   output [2:0] state;
   // end debug

   reg          hsyncbar, vsyncbar, blankbar;

   // modify the size of this var as necessary to be able to count the constants below
   reg [9:0]    cnt;
   reg [9:0]    vcnt;

   reg [2:0]    state, next;

   parameter    HEIGHT = 600;
   parameter    WIDTH = 800;

   parameter    H_FRONT_PORCH = 56;
   parameter    H_PULSE_LENGTH = 120;
   parameter    H_BACK_PORCH = 64;

   parameter    V_FRONT_PORCH_LENGTH = 37;
   parameter    V_PULSE_LENGTH = 6;
   parameter    V_BACK_PORCH_LENGTH = 23;

   // "line modes".  screen output (normal), vfp, vp, vbp
   parameter    LM_NORMAL = 0;
   parameter    LM_VFP = 1;
   parameter    LM_VP = 2;
   parameter    LM_VBP = 3;
   reg [1:0]    line_mode;
   reg          line_mode_change;

   parameter    IDLE = 0;
   parameter    HROW = 1;
   parameter    HFP = 2;
   parameter    HP = 3;
   parameter    HBP = 4;

   // Sequential always block for state assignment
   always @ (posedge clk or posedge reset) begin
      if (reset) begin
         state <= IDLE;
         cnt <= 1;
         vcnt <= 1;
         line_mode = LM_NORMAL;
      end
      else begin
         if (line_mode_change) begin
            line_mode = line_mode + 1; // blocking assign, we depend on this later... trust this to roll over correctly
         end

         hsyncbar <= !(next == HP);
         vsyncbar <= !(line_mode == LM_VP);
         if ((next == HFP) || (next == HP) || (next == HBP)) // horizontal blank
           blankbar <= 1'b0;
         else if ((line_mode == LM_VFP) || (line_mode == LM_VP) || (line_mode == LM_VBP))
           blankbar <= 1'b0;
         else
           blankbar <= 1'b1;

         if (next == IDLE)
           cnt <= 1;
         else if (state != next)
           cnt <= 1;
         else
           cnt <= cnt + 1;

         if (next == IDLE)
           vcnt <= 1;
         else if ((state == HBP) && (next == HROW)) begin
            if (line_mode_change)
              vcnt <= 1;
            else
              vcnt <= vcnt + 1;
         end
         else
           vcnt <= vcnt;

         state <= next;
      end // else: !if(reset)
   end // always @ (posedge clk or posedge reset)

   // Combinational always block for next-state
   // computation
   always @ (state or cnt) begin
      case (state)
        IDLE: begin
           next <= HROW;
           // line_mode <= LM_NORMAL;
           line_mode_change <= 1'b0;
        end

        HROW: begin
           line_mode_change <= 1'b0;
           if (cnt == WIDTH)
             next <= HFP;
           else
             next <= HROW;
        end

        HFP: begin
           if (cnt == H_FRONT_PORCH)
             next <= HP;
           else
             next <= HFP;
        end

        HP: begin
           if (cnt == H_PULSE_LENGTH)
             next <= HBP;
           else
             next <= HP;
        end

        HBP: begin
           if (cnt == H_BACK_PORCH) begin
              next <= HROW;
              // change the line mode if we've reached the limit for the current line mode
              if (line_mode == LM_NORMAL) begin
                 if (vcnt == HEIGHT) begin
                    // line_mode <= LM_VFP;
                    line_mode_change <= 1'b1;
                 end
              end
```

```
               else if (line_mode == LM_VFP) begin
                  if (vcnt == V_FRONT_PORCH_LENGTH) begin
                     // line_mode <= LM_VP;
                     line_mode_change <= 1'b1;
                  end
               end
               else if (line_mode == LM_VP) begin
                  if (vcnt == V_PULSE_LENGTH) begin
                     // line_mode <= LM_VBP;
                     line_mode_change <= 1'b1;
                  end
               end
               else if (line_mode == LM_VBP) begin
                  if (vcnt == V_BACK_PORCH_LENGTH) begin
                     // line_mode <= LM_NORMAL;
                     line_mode_change <= 1'b1;
                  end
               end
               else begin // shouldn't ever happen
                  $display("illegal condition!");
                  // line_mode <= LM_NORMAL;
               end
            end // if (cnt == H_BACK_PORCH)
            else
              next <= HBP;
         end // case: HBP
         default: begin
            next <= IDLE;
         end
       endcase
   end // always @ (state)
endmodule // sync_generator
```

```
module vram_display(clk, reset,
                    vram_read_data,
                    vga_out_hsync, vga_out_vsync, vga_out_blank_b, vga_out_sync_b, vga_out_red, vga_out_green, vga_out_blue,
                    vram_read_addr);

   input clk, reset;
   input [7:0] vram_read_data;
   output      vga_out_hsync, vga_out_vsync, vga_out_blank_b, vga_out_sync_b;
   output [7:0] vga_out_red, vga_out_green, vga_out_blue;
   output [15:0] vram_read_addr;

   wire          video_pixel;
   wire          vga_out_pixel_clock;

   assign        vga_out_pixel_clock = clk;
   assign        vga_out_red = video_pixel ? 8'b11111111 : 8'b00000000;
   assign        vga_out_blue = video_pixel ? 8'b11111111 : 8'b00000000;
   assign        vga_out_green = video_pixel ? 8'b11111111 : 8'b00000000;

   // sync generator
   wire          hsyncbar, vsyncbar, blank_b;
   wire [9:0]    hnum, vnum, hnum_norm, vnum_norm;
   reg [9:0]     hnum_norm_delay, vnum_norm_delay;
   // debug
   wire [2:0]    state;
   // end debug

   // instantiate the sync generator
   sync_generator sg(.clk(vga_out_pixel_clock),
                     .reset(reset),
                     .hsyncbar(hsyncbar),
                     .vsyncbar(vsyncbar),
                     .blankbar(blank_b),
                     .cnt(hnum),
                     .vcnt(vnum)
                     // debug
                     ,
                     .state(state)
                     // end debug
                     );
   // subtract 1
   assign hnum_norm = hnum + 10'b1111111111;
   assign vnum_norm = vnum + 19'b1111111111;

   assign        vram_read_addr = hnum_norm[9:3] + (vnum_norm * 100);
   wire [2:0]    current_bit;

   assign        current_bit = 3'd7 - hnum_norm_delay[2:0];
   assign        video_pixel = vram_read_data[current_bit];


   // compensate for ram and dac delay...
   // have to wait one clock period for ram data to show up
   // hsync and vsync are passed straight to the display, bypassing the dac, so delay them 2 clock cycles
   reg           hsyncbar_int1, vsyncbar_int1, hsyncbar_int2, vsyncbar_int2, hsyncbar_final, vsyncbar_final, vga_out_blank_b_final;
   always @ (posedge vga_out_pixel_clock) begin
      if (reset) begin
         hsyncbar_int1 <= 1'b0;
         vsyncbar_int1 <= 1'b0;
         hsyncbar_int2 <= 1'b0;
         vsyncbar_int2 <= 1'b0;
         hsyncbar_final <= 1'b0;
         vsyncbar_final <= 1'b0;
         vga_out_blank_b_final <= 1'b0;
         hnum_norm_delay <= 9'd0;
         vnum_norm_delay <= 9'd0;
      end
      else begin
         hsyncbar_int1 <= hsyncbar;
         vsyncbar_int1 <= vsyncbar;
         hsyncbar_int2 <= hsyncbar_int1;
         vsyncbar_int2 <= vsyncbar_int1;
         hsyncbar_final <= hsyncbar_int2;
         vsyncbar_final <= vsyncbar_int2;
         vga_out_blank_b_final <= blank_b;
         hnum_norm_delay <= hnum_norm;
         vnum_norm_delay <= vnum_norm;
      end
   end // always @ (posedge vga_out_pixel_clock)

   assign vga_out_hsync = hsyncbar_final;
   assign vga_out_vsync = vsyncbar_final;
   assign vga_out_sync_b = hsyncbar_int1 ^~ vsyncbar_int1;
   assign vga_out_blank_b = vga_out_blank_b_final;
endmodule // vram_display
```

```
module minor_fsm_raw_ckt(clk, reset, active, raw_ckt_dout,
                         raw_ckt_addr, vram_write_we, vram_write_addr, vram_write_data
// debug
                        ,
                        state,
                        vram_section_ctr
// end debug
);
   input clk, reset, active;
   input [63:0] raw_ckt_dout;

   output [11:0] raw_ckt_addr;
   output        vram_write_we;
   output [15:0] vram_write_addr;
   output [7:0]  vram_write_data;
   // debug
   output [3:0]  state;
   output [2:0]  vram_section_ctr;

   // end debug


   reg [11:0]    raw_ckt_addr;
   reg           vram_write_we;
   reg [15:0]    vram_write_addr;

   reg [2:0]     vram_section_ctr;

   reg [3:0]     state, next;
   reg           initializing;

   parameter     VERT_OFFSET = 44;
   parameter     HORIZ_OFFSET = 144;

   parameter     IDLE = 0;
   parameter     INITIALIZE = 1;
   parameter     VRAM_CLEAR_INIT_ADDR = 8;
   parameter     VRAM_CLEAR_SET_ADDR = 9;
   parameter     VRAM_CLEAR_WRITE_DATA = 10;
   parameter     VRAM_CLEAR_HOLD_DATA = 11;
   parameter     SET_RAW_ADDR = 2;
   parameter     READ_RAW_DATA = 3;
   parameter     SET_VRAM_ADDR = 4;
   parameter     WRITE_VRAM_DATA = 5;
   parameter     HOLD_VRAM_DATA = 6;
   parameter     TERMINATE = 7;

   assign        vram_write_data = initializing ? 8'hff : ~(raw_ckt_dout >> (8 * (3'd7 - vram_section_ctr)));

   always @ (posedge clk or posedge reset) begin
      if (reset)
        state <= IDLE;
      else begin
         if (next == INITIALIZE)
           initializing <= 1'b1;
         else if (next == SET_RAW_ADDR)
           initializing <= 1'b0;

         if (next == INITIALIZE)
           raw_ckt_addr <= 12'b111111111111;
         else if (next == SET_RAW_ADDR)
           raw_ckt_addr <= raw_ckt_addr + 1;

         if (next == INITIALIZE) begin
            vram_write_addr = 16'd0;
            vram_section_ctr = 3'b111;
         end
         else if (next == VRAM_CLEAR_INIT_ADDR) begin
            vram_write_addr = 16'd0;
            vram_section_ctr = 3'b111;
         end
         else if (next == VRAM_CLEAR_SET_ADDR) begin
            vram_write_addr = vram_write_addr + 1;
            vram_section_ctr = 3'b111;
         end
         else if (next == SET_VRAM_ADDR) begin
            vram_section_ctr = vram_section_ctr + 1;
            vram_write_addr = ((VERT_OFFSET * 100) +
                               (HORIZ_OFFSET >> 3) +
                               ((raw_ckt_addr >> 3) * 100) +
                               (raw_ckt_addr[2:0] << 3) +
                               vram_section_ctr);
         end

         if ((next == WRITE_VRAM_DATA) || (next == VRAM_CLEAR_WRITE_DATA))
           vram_write_we <= 1'b1;
         else
           vram_write_we <= 1'b0;

         state <= next;
      end // else: !if(reset)
   end // always @ (posedge clk or posedge reset)

   always @ (state or active) begin
      if (!active)
        next <= IDLE;
      else begin
         case (state)
           default: begin
              next <= IDLE;
           end

           IDLE: begin
              next <= INITIALIZE;
           end

           INITIALIZE: begin
              next <= VRAM_CLEAR_INIT_ADDR;
           end

           VRAM_CLEAR_INIT_ADDR: next <= VRAM_CLEAR_WRITE_DATA;
           VRAM_CLEAR_SET_ADDR: next <= VRAM_CLEAR_WRITE_DATA;
           VRAM_CLEAR_WRITE_DATA: next <= VRAM_CLEAR_HOLD_DATA;
           VRAM_CLEAR_HOLD_DATA: begin
              if (vram_write_addr == 16'd59999)
                next <= SET_RAW_ADDR;
              else
                next <= VRAM_CLEAR_SET_ADDR;
           end

           SET_RAW_ADDR: begin
              next <= READ_RAW_DATA;
           end

           READ_RAW_DATA: begin
              next <= SET_VRAM_ADDR;
           end

           SET_VRAM_ADDR: begin
              next <= WRITE_VRAM_DATA;
           end
```

```
            WRITE_VRAM_DATA: begin
               next <= HOLD_VRAM_DATA;
            end

            HOLD_VRAM_DATA: begin
               if (vram_section_ctr == 3'b111) begin // done with this component
                  if (raw_ckt_addr == 12'b111111111111) begin // circuit ctr is rolling over, we're done
                     next <= TERMINATE;
                  end
                  else begin
                     next <= SET_RAW_ADDR;
                  end
               end
               else begin // go to next section of component
                  next <= SET_VRAM_ADDR;
               end // else: !if(vram_section_ctr == 3'd0)
            end // case: HOLD_VRAM_DATA

            TERMINATE: begin
               next <= TERMINATE;
            end
         endcase // case(state)
      end // else: !if(!active)
   end // always @ (state or active)
endmodule // minor_fsm_ideal_ckt
```

```
module minor_fsm_ideal_ckt(clk, reset, active, ideal_data, comp_rom_data, char_rom_dout, node_data_out, nodefinder_finished,
                           ideal_addr, comp_rom_addr, char_rom_addr, node_addr, nodefinder_active, vram_write_we, vram_write_addr, vram_write_data
                           // debug
                           ,
                           state,
                           comp_section_ctr
                           // end debug
                           );
   input clk, reset, active;
   input [19:0] ideal_data;
   input [7:0]  comp_rom_data;
   input [7:0]  char_rom_dout;
   input [6:0]  node_data_out;
   input        nodefinder_finished;

   output [5:0] ideal_addr;
   output [13:0] comp_rom_addr;
   output [7:0]  char_rom_addr;
   output [6:0]  node_addr;
   output        nodefinder_active;
   output        vram_write_we;
   output [15:0] vram_write_addr;
   output [7:0]  vram_write_data;
   // debug
   output [5:0]  state;
   output [8:0]  comp_section_ctr;
   // end debug

   wire [4:0]    block_type;
   assign        block_type = ideal_data[19:15];

   reg [5:0]     state, next;

   reg [5:0]     ideal_addr;
   reg [13:0]    comp_rom_addr;
   reg [7:0]     char_rom_addr;
   reg           vram_write_we;
   reg [15:0]    vram_write_addr;
   reg [7:0]     vram_write_data;
   reg [6:0]     node_addr;
   reg           nodefinder_active;

   reg [8:0]     comp_section_ctr;
   reg [1:0]     digit_ctr;
   reg [2:0]     char_section_ctr;

   reg [1:0]     node_ctr;

   reg [2:0]     row_num, col_num;

   parameter     VERT_OFFSET = 44;
   parameter     HORIZ_OFFSET = 144;

   parameter     COMPONENT_THRESHOLD = 5'd12;
   parameter     NONEXISTENT_NODE = 7'd112;
   parameter     UNASSIGNED_NODE = 7'd113;

   parameter     IDLE = 0;
   parameter     INITIALIZE = 1;
   parameter     VRAM_CLEAR_INIT_ADDR = 29;
   parameter     VRAM_CLEAR_SET_ADDR = 30;
   parameter     VRAM_CLEAR_WRITE_DATA = 31;
   parameter     VRAM_CLEAR_HOLD_DATA = 32;
   parameter     ANALYZE = 27;
   parameter     INIT_CKT_ADDR = 28;
   parameter     SET_CKT_ADDR = 2;
   parameter     READ_CKT_DATA = 3;
   parameter     INIT_COMP_ADDR = 4;
   parameter     SET_COMP_ADDR = 5;
   parameter     READ_COMP_DATA = 6;
   parameter     SET_COMP_VRAM_ADDR = 7;
   parameter     WRITE_COMP_VRAM_DATA = 8;
   parameter     HOLD_COMP_VRAM_DATA = 9;
   parameter     INIT_CHAR_CTRS = 10;
   parameter     SET_CHAR_ADDR = 11;
   parameter     READ_CHAR_DATA = 12;
   parameter     SET_CHAR_VRAM_ADDR = 13;
   parameter     WRITE_CHAR_VRAM_DATA = 14;
   parameter     HOLD_CHAR_VRAM_DATA = 15;
   parameter     INIT_NODE_CTRS = 16;
   parameter     SET_NODE_ADDR_1 = 17;
   parameter     SKIP_NODE = 18;
   parameter     SET_NODE_ADDR_2 = 19;
   parameter     READ_NODE_DATA = 20;
   parameter     SET_NODE_CHAR_ADDR = 21;
   parameter     READ_NODE_CHAR_DATA = 22;
   parameter     SET_NODE_CHAR_VRAM_ADDR = 23;
   parameter     WRITE_NODE_CHAR_VRAM_DATA = 24;
   parameter     HOLD_NODE_CHAR_VRAM_DATA = 25;
   parameter     TERMINATE = 26;

   always @ (posedge clk or posedge reset) begin
      if (reset)
        state <= IDLE;
      else begin
         if (next == ANALYZE)
           nodefinder_active <= 1'b1;
         else
           nodefinder_active <= 1'b0;

         if (next == INIT_CKT_ADDR)
           ideal_addr <= 6'd0;
         else if (next == SET_CKT_ADDR)
           ideal_addr <= ideal_addr + 1;

         if (next == INITIALIZE)
           comp_section_ctr <= 9'd0;
         else if (next == HOLD_COMP_VRAM_DATA)
           comp_section_ctr <= comp_section_ctr + 1;

         if (next == INITIALIZE)
           comp_rom_addr <= 14'd0;
         else if ((next == INIT_COMP_ADDR) || (next == SET_COMP_ADDR))
           comp_rom_addr <= (block_type << 9) + comp_section_ctr;
         else
           comp_rom_addr <= comp_rom_addr;

         if (next == INITIALIZE)
           node_addr <= 7'bx;
         else if (next == SET_NODE_ADDR_1) begin
            case (node_ctr)
              2'b00: begin // top
                 // grid locs 0..7 do not have a top edge
                 if (ideal_addr < 6'd8)
                   node_addr <= NONEXISTENT_NODE;
                 else // 7 + 15*(row_num - 1) + col_num
                   node_addr <= (7 +
                                 (15 * (row_num - 1)) +
                                 col_num);
              end
              2'b01: begin // left
```

```
            // grid locs evenly divisible by 8 do not have a left edge
            if (col_num == 3'b000)
              node_addr <= NONEXISTENT_NODE;
            else // (15*row_num) + (location within row - 1)
              node_addr <= (15 * row_num) +
                           col_num - 1;
         end
         2'b10: begin // right
            // grid locs 1 short of being evenly divisible by 8 do not have a right edge
            if (col_num == 3'b111)
              node_addr <= NONEXISTENT_NODE;
            else // (15*row_num) + col_num
              node_addr <= (15 * row_num) +
                           col_num;
         end
         2'b11: begin // bottom
            // grid locs 56..63 do not have a bottom edge
            if (ideal_addr > 6'd55)
              node_addr <= NONEXISTENT_NODE;
            else // 7 + (15*row_num) + col_num
              node_addr <= (7 +
                            (15 * row_num) +
                            col_num);
         end
      endcase // case(node_ctr)
   end // if (next == SET_NODE_ADDR_1)

   if (next == VRAM_CLEAR_INIT_ADDR)
     vram_write_data <= 8'hff;
   else if (next == SET_COMP_VRAM_ADDR)
     vram_write_data <= comp_rom_data;
   else if (next == SET_CHAR_VRAM_ADDR)
     vram_write_data <= char_rom_dout;
   else if (next == SET_NODE_CHAR_VRAM_ADDR)
     vram_write_data <= char_rom_dout;


   if (next == VRAM_CLEAR_INIT_ADDR)
     vram_write_addr <= 16'd0;
   else if (next == VRAM_CLEAR_SET_ADDR)
     vram_write_addr <= vram_write_addr + 1;
   else if (next == SET_COMP_VRAM_ADDR)
     vram_write_addr <= ((VERT_OFFSET * 100) +
                         (HORIZ_OFFSET >> 3) +
                         (ideal_addr[5:3] * 6400) + // initial vertical offset for entire block
                         (ideal_addr[2:0] * 8) + // initial horizontal offset for entire block
                         (100 * comp_section_ctr[8:3]) + // vertical offset within component block
                         comp_section_ctr[2:0]); // horizontal offset within component block
   else if (next == SET_CHAR_VRAM_ADDR) begin
      if (digit_ctr == 2'b11) // multiplier
        vram_write_addr <= ((VERT_OFFSET * 100) +
                            (HORIZ_OFFSET >> 3) +
                            (ideal_addr[5:3] * 6400) + // initial vertical offset for entire block
                            (ideal_addr[2:0] * 8) + // initial horizontal offset for entire block
                            (5600 + (100 * char_section_ctr)) + // vertical offset within component block
                            7); // horizontal offset within component block
      else // digit
        vram_write_addr <= ((VERT_OFFSET * 100) +
                            (HORIZ_OFFSET >> 3) +
                            (ideal_addr[5:3] * 6400) + // initial vertical offset for entire block
                            (ideal_addr[2:0] * 8) + // initial horizontal offset for entire block
                            (4800 + (100 * char_section_ctr)) + // vertical offset within component block
                            5 + digit_ctr); // horizontal offset within component block
   end // if (next == SET_CHAR_VRAM_ADDR)
   else if (next == SET_NODE_CHAR_VRAM_ADDR) begin
      case (node_ctr)
        2'b00: // top node, starts at 2,-1
          vram_write_addr <= ((VERT_OFFSET * 100) +
                              (HORIZ_OFFSET >> 3) +
                              (ideal_addr[5:3] * 6400) + //initial vertical offset for entire block
                              (ideal_addr[2:0] * 8) +  // initial horizontal offset for entire block
                              (100 * char_section_ctr) + // vertical offset within component block
                              2 + digit_ctr - 800); // horizontal offset within component block
        2'b01: // left node, starts at -2,3
          vram_write_addr <= ((VERT_OFFSET * 100) +
                              (HORIZ_OFFSET >> 3) +
                              (ideal_addr[5:3] * 6400) + //initial vertical offset for entire block
                              (ideal_addr[2:0] * 8) +  // initial horizontal offset for entire block
                              (2400 + (100 * char_section_ctr)) + // vertical offset within component block
                              digit_ctr - 402); // horizontal offset within component block
        2'b10: // right node, starts at 6,3
          vram_write_addr <= ((VERT_OFFSET * 100) +
                              (HORIZ_OFFSET >> 3) +
                              (ideal_addr[5:3] * 6400) + //initial vertical offset for entire block
                              (ideal_addr[2:0] * 8) +  // initial horizontal offset for entire block
                              (2400 + (100 * char_section_ctr)) + // vertical offset within component block
                              6 + digit_ctr - 400); // horizontal offset within component block
        2'b11: // bottom node, starts at 2,7
          vram_write_addr <= ((VERT_OFFSET * 100) +
                              (HORIZ_OFFSET >> 3) +
                              (ideal_addr[5:3] * 6400) + //initial vertical offset for entire block
                              (ideal_addr[2:0] * 8) +  // initial horizontal offset for entire block
                              (5600 + (100 * char_section_ctr)) + // vertical offset within component block
                              2 + digit_ctr); // horizontal offset within component block
      endcase // case(node_ctr)
   end // if (next == SET_NODE_CHAR_VRAM_ADDR)

   if ((next == WRITE_COMP_VRAM_DATA) || (next == WRITE_CHAR_VRAM_DATA) || (next == WRITE_NODE_CHAR_VRAM_DATA) || (next == VRAM_CLEAR_WRITE_DATA))
     vram_write_we <= 1'b1;
   else
     vram_write_we <= 1'b0;

   if (next == INIT_CHAR_CTRS) begin
      char_section_ctr <= 3'd0;
      digit_ctr <= 2'd0;
      node_ctr <= 2'bxx;
      row_num <= 3'bxxx;
      col_num <= 3'bxxx;
   end
   else if (next == INIT_NODE_CTRS) begin
      char_section_ctr <= 3'd0;
      digit_ctr <= 2'd0;
      node_ctr <= 2'd0;
      row_num <= ideal_addr[5:3];
      col_num <= ideal_addr[2:0];
   end
   else if (next == HOLD_CHAR_VRAM_DATA) begin
      char_section_ctr <= char_section_ctr + 1;
      if (char_section_ctr == 3'd7)
        digit_ctr <= digit_ctr + 1;
      node_ctr <= 2'bxx;
      row_num <= 3'bxxx;
      col_num <= 3'bxxx;
   end
   else if (next == HOLD_NODE_CHAR_VRAM_DATA) begin
      char_section_ctr <= char_section_ctr + 1;
      if (char_section_ctr == 3'd7) begin
         if (digit_ctr == 2'd1) begin
            digit_ctr <= 2'd0;
            node_ctr <= node_ctr + 1;
```

```
            end
            else begin
               digit_ctr <= digit_ctr + 1;
               node_ctr <= node_ctr;
            end
          end
          row_num <= row_num;
          col_num <= col_num;
       end // if (next == HOLD_NODE_CHAR_VRAM_DATA)
       else if (next == SKIP_NODE)
         node_ctr <= node_ctr + 1;

       if (next == SET_CHAR_ADDR) begin
          case (digit_ctr)
            2'b00: // digit 1
              if (ideal_data[14:11] == 4'b1111) // blank
                char_rom_addr <= 8'd248 + char_section_ctr;
              else
                char_rom_addr <= (ideal_data[14:11] << 3) + char_section_ctr;
            2'b01: //digit 2
              if (ideal_data[10:7] == 4'b1111) // blank
                char_rom_addr <= 8'd248 + char_section_ctr;
              else
                char_rom_addr <= (ideal_data[10:7] << 3) + char_section_ctr;
            2'b10: // digit 3
              if (ideal_data[6:3] == 4'b1111) // blank
                char_rom_addr <= 8'd248 + char_section_ctr;
              else
                char_rom_addr <= (ideal_data[6:3] << 3) + char_section_ctr;
            2'b11: begin // multiplier
               if (ideal_data[2:0] == 3'd7) // blank
                 char_rom_addr <= 8'd248 + char_section_ctr;
               else // ((16 + mult_code) * 8) + offset
                 char_rom_addr <= ((16 + ideal_data[2:0]) << 3) + char_section_ctr;
            end
            default: char_rom_addr <= 8'bx; // shouldn't happen
          endcase // case(digit_ctr)
       end // if (next == SET_CHAR_ADDR)
       else if (next == SET_NODE_CHAR_ADDR) begin
          case (digit_ctr) // octal display
            2'b00: begin // first digit
               if (node_data_out < 8) // don't display leading 0
                 char_rom_addr <= 8'd248 + char_section_ctr;
               else
                 char_rom_addr <= {node_data_out[5:3], char_section_ctr};
            end // case: 2'b00
            2'b01: char_rom_addr <= {node_data_out[2:0], char_section_ctr}; // 2nd digit
          endcase // case(digit_ctr)
       end // if (next == SET_NODE_CHAR_ADDR)

       state <= next;
    end // else: !if(reset)
 end // always @ (posedge clk or posedge reset)

 always @ (state or active or comp_rom_data or char_rom_dout or nodefinder_finished) begin
    if (!active)
      next <= IDLE;
    else begin
       case (state)
         default: begin
            next <= IDLE;
         end

         IDLE: begin
            next <= INITIALIZE;
         end

         INITIALIZE: begin
            next <= VRAM_CLEAR_INIT_ADDR;
         end

         VRAM_CLEAR_INIT_ADDR: begin
            next <= VRAM_CLEAR_WRITE_DATA;
         end

         VRAM_CLEAR_SET_ADDR: begin
            next <= VRAM_CLEAR_WRITE_DATA;
         end

         VRAM_CLEAR_WRITE_DATA: begin
            next <= VRAM_CLEAR_HOLD_DATA;
         end

         VRAM_CLEAR_HOLD_DATA: begin
            if (vram_write_addr == 16'd59999)
              next <= ANALYZE;
            else
              next <= VRAM_CLEAR_SET_ADDR;
         end

         ANALYZE: begin
            if (nodefinder_finished)
              next <= INIT_CKT_ADDR;
            else
              next <= ANALYZE;
         end

         INIT_CKT_ADDR: next <= READ_CKT_DATA;

         SET_CKT_ADDR: begin
            next <= READ_CKT_DATA;
         end

         READ_CKT_DATA: begin
            next <= INIT_COMP_ADDR;
         end

         INIT_COMP_ADDR: begin
            next <= READ_COMP_DATA;
         end

         SET_COMP_ADDR: begin
            next <= READ_COMP_DATA;
         end

         READ_COMP_DATA: begin
            next <= SET_COMP_VRAM_ADDR;
         end

         SET_COMP_VRAM_ADDR: begin
            next <= WRITE_COMP_VRAM_DATA;
         end

         WRITE_COMP_VRAM_DATA: begin
            next <= HOLD_COMP_VRAM_DATA;
         end

         HOLD_COMP_VRAM_DATA: begin
            if (comp_section_ctr == 9'd0) // done with all sections of component, move on to characters
              next <= INIT_CHAR_CTRS;
```

```
               else // go to next section of component
                 next <= SET_COMP_ADDR;
            end

            INIT_CHAR_CTRS: next <= SET_CHAR_ADDR;

            SET_CHAR_ADDR: begin
               next <= READ_CHAR_DATA;
            end

            READ_CHAR_DATA: begin
               next <= SET_CHAR_VRAM_ADDR;
            end

            SET_CHAR_VRAM_ADDR: begin
               next <= WRITE_CHAR_VRAM_DATA;
            end

            WRITE_CHAR_VRAM_DATA: begin
               next <= HOLD_CHAR_VRAM_DATA;
            end

            HOLD_CHAR_VRAM_DATA: begin
               if ((digit_ctr == 2'd0) && (char_section_ctr == 3'd0)) begin // last piece of last character written, move on to writing node info if we h
ave to
                  if (block_type < COMPONENT_THRESHOLD)
                    next <= INIT_NODE_CTRS;
                  else begin // move on to next component, if there is one
                     if (ideal_addr == 6'b111111) // no more components, terminate
                       next <= TERMINATE;
                     else
                       next <= SET_CKT_ADDR;
                  end
               end
               else
                 next <= SET_CHAR_ADDR;
            end // case: HOLD_CHAR_VRAM_DATA
            INIT_NODE_CTRS: next <= SET_NODE_ADDR_1;
            SET_NODE_ADDR_1: begin
               if (node_addr == NONEXISTENT_NODE)
                 next <= SKIP_NODE;
               else
                 next <= SET_NODE_ADDR_2;
            end

            SKIP_NODE: begin
               if (node_ctr == 2'd0) begin // done with all nodes
                  if (ideal_addr == 6'b111111) // done with all components
                    next <= TERMINATE;
                  else // move on to next component
                    next <= SET_CKT_ADDR;
               end
               else
                 next <= SET_NODE_ADDR_1;
            end

            SET_NODE_ADDR_2: next <= READ_NODE_DATA;
            READ_NODE_DATA: begin
               if (node_data_out == UNASSIGNED_NODE)
                 next <= SKIP_NODE;
               else
                 next <= SET_NODE_CHAR_ADDR;
            end

            SET_NODE_CHAR_ADDR: next <= READ_NODE_CHAR_DATA;
            READ_NODE_CHAR_DATA: begin
               next <= SET_NODE_CHAR_VRAM_ADDR;
            end

            SET_NODE_CHAR_VRAM_ADDR: begin
               next <= WRITE_NODE_CHAR_VRAM_DATA;
            end

            WRITE_NODE_CHAR_VRAM_DATA: begin
               next <= HOLD_NODE_CHAR_VRAM_DATA;
            end

            HOLD_NODE_CHAR_VRAM_DATA: begin
               if ((digit_ctr == 2'd0) && (char_section_ctr == 3'd0)) begin // done writing this node
                  if (node_ctr == 2'd0) begin // done with all nodes
                     if (ideal_addr == 6'b111111) // done with all components
                       next <= TERMINATE;
                     else // move on to next component
                       next <= SET_CKT_ADDR;
                  end
                  else // move on to next node
                    next <= SET_NODE_ADDR_1;
//                  next <= TERMINATE;
               end
               else // move on to next part of digit or next digit
                 next <= SET_NODE_CHAR_ADDR;
            end // case: HOLD_NODE_CHAR_VRAM_DATA

            TERMINATE: begin
               next <= TERMINATE;
            end
          endcase // case(state)
       end // else: !if(!active)
   end // always @ (state)
endmodule // minor_fsm_ideal_ckt
```

```
module minor_fsm_spice(clk, reset, active, ideal_data, node_data_out, nodefinder_finished, spice_data_read, char_rom_dout, serial_export,
                       ideal_addr, node_addr, nodefinder_active, vram_write_we, vram_write_addr, vram_write_data, spice_we, spice_addr, spice_data_write
, char_rom_addr, rs232_txd, rs232_rts
                       // debug
                       ,
                       state
                       // end debug
                       );
   input clk, reset, active;
   input [19:0] ideal_data;
   input [6:0]  node_data_out;
   input        nodefinder_finished;
   input [4:0]  spice_data_read;
   input [7:0]  char_rom_dout;
   input        serial_export;

   output [5:0] ideal_addr;
   output [6:0] node_addr;
   output       nodefinder_active;
   output       vram_write_we;
   output [15:0] vram_write_addr;
   output [7:0]  vram_write_data;
   output        spice_we;
   output [10:0] spice_addr;
   output [4:0]  spice_data_write;
   output [7:0]  char_rom_addr;
   output        rs232_txd;
   output        rs232_rts;

   // debug
   output [5:0]  state;
   // end debug


   // bigchar_rom_24x768
   wire [23:0]   bigchar_rom_dout;
   reg [9:0]     bigchar_rom_addr;
   reg [4:0]     bigchar_section_ctr;
   reg [1:0]     vram_section_ctr;
   bigchar_rom_24x768 bigchar_rom(.clk(clk),
                                  .addr(bigchar_rom_addr),
                                  .dout(bigchar_rom_dout));



   reg [5:0]     ideal_addr;
   reg [6:0]     node_addr;
   reg           nodefinder_active, vram_write_we;
   reg [15:0]    vram_write_addr;
   reg [7:0]     vram_write_data;
   reg           spice_we;
   wire [10:0]   spice_addr;
   reg [10:0]    private_spice_addr;
   reg [4:0]     spice_data_write;
   reg [7:0]     char_rom_addr;

   reg [5:0]     state, next;

   parameter     VERT_OFFSET = 0; // rows (pixels)
   parameter     HORIZ_OFFSET = 24; // pixels, must be multiple of 8
   parameter     INTERLINE_SPACE = 5; // rows (pixels)


   parameter     COMPONENT_THRESHOLD = 5'd12;
   parameter     NONEXISTENT_NODE = 7'd112;
   parameter     CHR_EOF = 5'd30;
   parameter     CHR_SPACE = 5'd31;
   parameter     CHR_NEWLINE = 5'd23;

   parameter     IDLE = 0;
   parameter     INITIALIZE = 1;
   parameter     ANALYZE = 2;
   parameter     SET_IDEAL_ADDR = 3;
   parameter     READ_IDEAL_DATA = 4;
   parameter     WRITE_ID_INIT = 5;
   parameter     WRITE_ID_SET_SPICE_ADDR = 6;
   parameter     WRITE_ID_WRITE_SPICE_DATA = 7;
   parameter     WRITE_ID_HOLD_SPICE_DATA = 8;
   parameter     SET_NODE_ADDR = 9;
   parameter     READ_NODE_DATA = 10;
   parameter     WRITE_NODE_SET_SPICE_ADDR = 11;
   parameter     WRITE_NODE_WRITE_SPICE_DATA = 12;
   parameter     WRITE_NODE_HOLD_SPICE_DATA = 13;
   parameter     WRITE_TYPE_SET_SPICE_ADDR = 14;
   parameter     WRITE_TYPE_WRITE_SPICE_DATA = 15;
   parameter     WRITE_TYPE_HOLD_SPICE_DATA = 16;
   parameter     WRITE_VALUE_SET_SPICE_ADDR = 17;
   parameter     WRITE_VALUE_WRITE_SPICE_DATA = 18;
   parameter     WRITE_VALUE_HOLD_SPICE_DATA = 19;
   parameter     WRITE_CR_SET_SPICE_ADDR = 20;
   parameter     WRITE_CR_WRITE_SPICE_DATA = 21;
   parameter     WRITE_CR_HOLD_SPICE_DATA = 22;
   parameter     WRITE_TERM_SET_SPICE_ADDR = 23;
   parameter     WRITE_TERM_WRITE_SPICE_DATA = 24;
   parameter     WRITE_TERM_HOLD_SPICE_DATA = 25;
   parameter     VRAM_CLEAR_INIT_ADDR = 26;
   parameter     VRAM_CLEAR_SET_ADDR = 27;
   parameter     VRAM_CLEAR_WRITE_DATA = 28;
   parameter     VRAM_CLEAR_HOLD_DATA = 29;
   parameter     VRAM_INIT_SPICE_ADDR = 30;
   parameter     VRAM_SET_SPICE_ADDR = 31;
   parameter     VRAM_READ_SPICE_DATA = 32;
   parameter     VRAM_SET_CHAR_ADDR = 33;
   parameter     VRAM_READ_CHAR_DATA = 34;
   parameter     VRAM_SET_VRAM_ADDR = 35;
   parameter     VRAM_WRITE_VRAM_DATA = 36;
   parameter     VRAM_HOLD_VRAM_DATA = 37;
   parameter     TERMINATE = 38;

   parameter     C_BLANK = 0;
   parameter     C_PSUPBOT = 1;
   parameter     C_PSUPTOP = 2;
   parameter     C_GRND = 3;
   parameter     C_VRES = 4;
   parameter     C_HRES = 5;
   parameter     C_VCAP = 6;
   parameter     C_HCAP = 7;
   parameter     C_VSRC = 8;
   parameter     C_HSRC = 9;
   parameter     C_LTRANS = 10;
   parameter     C_RTRANS = 11;

   wire          master_serial_export;
   assign        master_serial_export = (state == TERMINATE) ? serial_export : 1'b0;

   reg [4:0]     vsrc_ctr, res_ctr, trans_ctr, cap_ctr, current_label_char, current_label_num;
   wire [4:0]    block_type;
   assign        block_type = ideal_data[19:15];
   reg [1:0]     num_terminals;
   reg [1:0]     terminal_ctr;
   reg           skip;
```

```
   reg [2:0]     col_num, row_num;
   reg [1:0]     write_id_ctr; // first=type,second=first id digit,third=second id digit, fourth=space
   reg [1:0]     write_node_ctr; // first=first digit, second=second digit, third=space
   reg [1:0]     write_type_ctr; // first=first char, second=second char, third=third char, fourth = space
   reg [1:0]     write_value_ctr; // first=first dig,second=second dig,third=third dig, fourth=mult

   reg [5:0]     line_number;
   reg [4:0]     char_number;

   // ascii rom
   wire [7:0]    ascii_out;

   asciirom_8x32 my_asciirom(.addr(spice_data_read),
                             .clk(clk),
                             .dout(ascii_out));

   // serial exporter
   wire          serial_fsm_finished;
   wire [10:0]   serial_spice_addr;

   serial_fsm my_serial_fsm(.clk(clk),
                            .reset_sync(reset),
                            .start(master_serial_export),
                            .finished(serial_fsm_finished),
                            .ascii_out(ascii_out),
                            .char_add(serial_spice_addr),
                            .char_out(spice_data_read),
                            .TxD(rs232_txd),
                            .TxD_busy(rs232_rts));

   assign        spice_addr = (state == TERMINATE) ? serial_spice_addr : private_spice_addr;

   always @ (posedge clk or posedge reset) begin
      if (reset)
        state <= IDLE;
      else begin
         // nodefinder control
         if (next == ANALYZE)
           nodefinder_active <= 1'b1;
         else
           nodefinder_active <= 1'b0;

         // spice ram write enable
         if ((next == WRITE_ID_WRITE_SPICE_DATA) || (next == WRITE_NODE_WRITE_SPICE_DATA) || (next == WRITE_TYPE_WRITE_SPICE_DATA) || (next == WRITE_VAL
UE_WRITE_SPICE_DATA) || (next == WRITE_CR_WRITE_SPICE_DATA) || (next == WRITE_TERM_WRITE_SPICE_DATA))
           spice_we <= 1'b1;
         else
           spice_we <= 1'b0;

         // ideal address
         if (next == INITIALIZE)
           ideal_addr <= 6'd0;
         else if (next == SET_IDEAL_ADDR)
           ideal_addr <= ideal_addr + 1;

         // row and col #
         if (next == READ_IDEAL_DATA) begin
            row_num <= ideal_addr[5:3];
            col_num <= ideal_addr[2:0];
         end

         if (next == INITIALIZE) begin
            skip <= 1'b0;
            vsrc_ctr <= 5'd0;
            res_ctr <= 5'd0;
            cap_ctr <= 5'd0;
            trans_ctr <= 5'd0;
            current_label_char <= 5'd0;
            current_label_num <= 5'd0;
            num_terminals <= 2'd0;
         end
         else if (next == WRITE_ID_INIT) begin
            if (block_type < COMPONENT_THRESHOLD) begin
               case (block_type)
                 C_BLANK, C_GRND:  skip <= 1'b1;
                 C_PSUPBOT, C_PSUPTOP, C_VSRC, C_HSRC: begin // voltage sources
                    skip <= 1'b0;
                    num_terminals <= 2'd3; //2
                    current_label_char <= 5'd10; // V
                    current_label_num <= vsrc_ctr;
                    $write("V%d", vsrc_ctr);
                    vsrc_ctr <= vsrc_ctr + 1;
                 end
                 C_VRES, C_HRES: begin // resistors
                    skip <= 1'b0;
                    num_terminals <= 2'd3; //2
                    current_label_char <= 5'd12; // R
                    current_label_num <= res_ctr;
                    $write("R%d", res_ctr);
                    res_ctr <= res_ctr + 1;
                 end
                 C_VCAP, C_HCAP: begin // capacitors
                    skip <= 1'b0;
                    num_terminals <= 2'd3; //2
                    current_label_char <= 5'd11; // C
                    current_label_num <= cap_ctr;
                    $write("C%d", cap_ctr);
                    cap_ctr <= cap_ctr + 1;
                 end
                 C_LTRANS, C_RTRANS: begin // transistors
                    skip <= 1'b0;
                    num_terminals <= 2'd3;
                    current_label_char <= 5'd13; // Q
                    current_label_num <= trans_ctr;
                    $write("Q%d", trans_ctr);
                    trans_ctr <= trans_ctr + 1;
                 end
                 default: $display("ERROR! UNANTICIPATED CASE");
               endcase // case(block_type)
            end // if (block_type < COMPONENT_THRESHOLD)
            else begin
               skip <= 1'b1;
            end
         end // if (next == WRITE_ID_INIT)

         if ((next == WRITE_ID_SET_SPICE_ADDR) || (next == WRITE_NODE_SET_SPICE_ADDR) || (next == WRITE_TYPE_SET_SPICE_ADDR) || (next == WRITE_VALUE_SET
_SPICE_ADDR) || (next == WRITE_CR_SET_SPICE_ADDR) || (next == WRITE_TERM_SET_SPICE_ADDR) || (next == VRAM_SET_SPICE_ADDR))
           private_spice_addr <= private_spice_addr + 1;
         else if (next == INITIALIZE)
           private_spice_addr <= 11'd2047;
         else if (next == VRAM_INIT_SPICE_ADDR)
           private_spice_addr <= 11'd0;

         if (next == WRITE_ID_SET_SPICE_ADDR) begin
            case (write_id_ctr)
              2'b00: spice_data_write <= current_label_char;
              2'b01: spice_data_write <= (current_label_num >> 3);
              2'b10: spice_data_write <= current_label_num[2:0];
              2'b11: spice_data_write <= CHR_SPACE;
            endcase // case(write_id_ctr)
```

```
            write_id_ctr <= write_id_ctr + 1;
        end
        else if (next == WRITE_NODE_SET_SPICE_ADDR) begin
            if (node_addr == NONEXISTENT_NODE) begin
                if ((terminal_ctr == 2'b01) && ((block_type == C_PSUPTOP) || (block_type == C_PSUPBOT))) begin
                    case (write_node_ctr)
                      2'b00, 2'b10: spice_data_write <= CHR_SPACE;
                      2'b01: spice_data_write <= 0;
                      2'b11: spice_data_write <= 5'bxxxxx;
                    endcase // case(write_node_ctr)
                end
                else
                  spice_data_write <= CHR_SPACE;
            end // if (node_addr == NONEXISTENT_NODE)
            else begin
                case (write_node_ctr) // octal display
                  2'b00: begin // don't print leading zeros
                      if (node_data_out[5:3] != 3'b000)
                        spice_data_write <= (node_data_out >> 3);
                      else
                        spice_data_write <= CHR_SPACE;
                  end
                  2'b01: spice_data_write <= node_data_out[2:0];
                  2'b10: spice_data_write <= CHR_SPACE;
                  2'b11: spice_data_write <= 5'bxxxxx;
                endcase // case(write_node_ctr)
            end
            if (write_node_ctr == 2'b10) begin
               write_node_ctr <= 2'b00;
               terminal_ctr <= terminal_ctr + 1;
            end
            else
              write_node_ctr <= write_node_ctr + 1;
        end // if (next == WRITE_NODE_SET_SPICE_ADDR)
        else if (next == WRITE_TYPE_SET_SPICE_ADDR) begin
            case (write_type_ctr)
              2'b00: begin
                  if ((block_type == C_LTRANS) || (block_type == C_RTRANS))
                    spice_data_write <= 14; // N
                  else if ((block_type == C_VSRC) || (block_type == C_HSRC))
                    spice_data_write <= 24; // D
                  else
                    spice_data_write <= CHR_SPACE;
              end
              2'b01: begin
                  if ((block_type == C_LTRANS) || (block_type == C_RTRANS))
                    spice_data_write <= 15; // P
                  else if ((block_type == C_VSRC) || (block_type == C_HSRC))
                    spice_data_write <= 11; // C
                  else
                    spice_data_write <= CHR_SPACE;
              end
              2'b10: begin
                  if ((block_type == C_LTRANS) || (block_type == C_RTRANS))
                    spice_data_write <= 14; // N
                  else
                    spice_data_write <= CHR_SPACE;
              end
              2'b11: spice_data_write <= CHR_SPACE;
            endcase // case(write_type_ctr)
            write_type_ctr <= write_type_ctr + 1;
        end // if (next == WRITE_TYPE_SET_SPICE_ADDR)
        else if (next == WRITE_VALUE_SET_SPICE_ADDR) begin
            case (write_value_ctr)
              2'b00: begin
                  if (ideal_data[14:11] == 4'b1111)
                    spice_data_write <= CHR_SPACE;
                  else
                    spice_data_write <= ideal_data[14:11];
              end
              2'b01: begin
                  if (ideal_data[10:7] == 4'b1111)
                    spice_data_write <= CHR_SPACE;
                  else
                    spice_data_write <= ideal_data[10:7];
              end
              2'b10: begin
                  if (ideal_data[6:3] == 4'b1111)
                    spice_data_write <= CHR_SPACE;
                  else
                    spice_data_write <= ideal_data[6:3];
              end
              2'b11: begin // multiplier
                  if (ideal_data[2:0] == 3'b111)
                    spice_data_write <= CHR_SPACE;
                  else
                    spice_data_write <= 16 + ideal_data[2:0];
              end
            endcase // case(write_value_ctr)
            write_value_ctr <= write_value_ctr + 1;
        end // if (next == WRITE_VALUE_SET_SPICE_ADDR)
        else if (next == WRITE_CR_SET_SPICE_ADDR)
          spice_data_write <= CHR_NEWLINE;
        else if (next == WRITE_TERM_SET_SPICE_ADDR)
          spice_data_write <= CHR_EOF;
        else if (next == WRITE_ID_INIT) begin
            write_node_ctr <= 2'd0;
            terminal_ctr <= 2'd0;
            write_id_ctr <= 2'd0;
            write_type_ctr <= 2'd0;
            write_value_ctr <= 2'd0;
        end

        // node addr
        if (next == SET_NODE_ADDR) begin
            case (terminal_ctr)
              2'b00: begin // first terminal
                  case (block_type)
                    C_PSUPBOT : begin // first node is bottom
                        node_addr <= (7 +
                                      (15 * row_num) +
                                      col_num);
                    end
                    C_PSUPTOP, C_VRES, C_VCAP, C_VSRC, C_LTRANS, C_RTRANS: begin // first node is top
                        node_addr <= (7 +
                                      (15 * (row_num - 1)) +
                                      col_num);
                    end
                    C_HRES, C_HCAP, C_HSRC: begin // first node is right
                        node_addr <= (15 * row_num) +
                                     col_num;
                    end
                    default: $display("ERROR! found a component with unspecified first node");
                  endcase // case(block_type)
              end // case: 2'b00
              2'b01: begin // second terminal
                  case (block_type)
                    C_PSUPBOT, C_PSUPTOP: begin // second node is hardcoded 0
                        node_addr <= NONEXISTENT_NODE;
                    end
```

```
                C_VRES, C_VCAP, C_VSRC: begin // second node is bottom
                   node_addr <= (7 +
                                 (15 * row_num) +
                                 col_num);
                end
                C_HRES, C_HCAP, C_HSRC, C_LTRANS: begin // second node is left
                   node_addr <= (15 * row_num) +
                                col_num - 1;
                end
                C_RTRANS: begin // second node is right
                   node_addr <= (15 * row_num) +
                                col_num;
                end
                default : $display("ERROR! UNANTICIPATED 2nd TERMINAL");
              endcase // case(block_type)
            end // case: 2'b01
            2'b10: begin // third terminal
               case (block_type)
                 C_LTRANS, C_RTRANS: begin // bottom
                   node_addr <= (7 +
                                 (15 * row_num) +
                                 col_num);
                end
                default: node_addr <= NONEXISTENT_NODE; // only 2 3-terminal devices
              endcase // case(block_type)
            end
          endcase // case(terminal_ctr)
        end // if (next == SET_NODE_ADDR)

        // videoram stuff
        if ((next == VRAM_CLEAR_WRITE_DATA) || (next == VRAM_WRITE_VRAM_DATA))
          vram_write_we <= 1'b1;
        else
          vram_write_we <= 1'b0;

        if (next == VRAM_CLEAR_INIT_ADDR)
          vram_write_data <= 8'hff;
        else if (next == VRAM_SET_VRAM_ADDR)
          case (vram_section_ctr)
            2'd0: vram_write_data <= bigchar_rom_dout[23:16];
            2'd1: vram_write_data <= bigchar_rom_dout[15:8];
            2'd2: vram_write_data <= bigchar_rom_dout[7:0];
            default: vram_write_data <= 16'bxxxxxxxxxxxxxxxx;
          endcase // case(bigchar_section_ctr)

        if (next == VRAM_SET_CHAR_ADDR)
          bigchar_rom_addr <= (spice_data_read * 24) + bigchar_section_ctr;

        if (next == VRAM_INIT_SPICE_ADDR) begin
           line_number <= 6'd0;
           char_number <= 5'd0;
           vram_write_addr <= 16'd0;
        end
        else if (next == VRAM_CLEAR_INIT_ADDR)
           vram_write_addr <= 16'd0;
        else if (next == VRAM_CLEAR_SET_ADDR)
           vram_write_addr <= vram_write_addr + 1;
        else if (next == VRAM_SET_VRAM_ADDR) begin
           vram_write_addr <= ((VERT_OFFSET * 100) +
                               (HORIZ_OFFSET >> 3) +
                               (line_number * 2400) +
                               (INTERLINE_SPACE * line_number * 100) +
                               (char_number * 3) + vram_section_ctr +
                               (bigchar_section_ctr * 100));
           if (spice_data_read == CHR_NEWLINE) begin
              line_number <= line_number + 1;
              char_number <= 5'd0;
           end
           else begin
              if (bigchar_section_ctr == 5'd23)
                char_number <= char_number + 1;
           end
        end // if (next == VRAM_SET_VRAM_ADDR)

        if ((next == VRAM_INIT_SPICE_ADDR) || (next == VRAM_SET_SPICE_ADDR)) begin
           bigchar_section_ctr <= 5'd0;
           vram_section_ctr <= 2'd0;
        end
        else if (next == VRAM_HOLD_VRAM_DATA) begin
           if (vram_section_ctr == 2'd2) begin
              vram_section_ctr <= 2'd0;
              if (bigchar_section_ctr == 5'd23)
                bigchar_section_ctr <= 5'd0;
              else
                bigchar_section_ctr <= bigchar_section_ctr + 1;
           end
           else
             vram_section_ctr <= vram_section_ctr + 1;
        end // if (next == VRAM_HOLD_VRAM_DATA)

        state <= next;
     end
   end

   always @ (state or active or nodefinder_finished) begin
      if (!active)
        next <= IDLE;
      else begin
         case (state)
           IDLE: next <= INITIALIZE;
           INITIALIZE: next <= ANALYZE;
           ANALYZE: begin
              if (nodefinder_finished)
                next <= READ_IDEAL_DATA;
              else
                next <= ANALYZE;
           end
           SET_IDEAL_ADDR: next <= READ_IDEAL_DATA;
           READ_IDEAL_DATA: next <= WRITE_ID_INIT;
           WRITE_ID_INIT: begin
              if (skip == 1'b0)
                next <= WRITE_ID_SET_SPICE_ADDR;
              else begin
                 if (ideal_addr == 6'd63)
                   next <= WRITE_CR_SET_SPICE_ADDR; // TODO: change? gives us 2crs at eof
                 else
                   next <= SET_IDEAL_ADDR;
              end
           end
           WRITE_ID_SET_SPICE_ADDR: next <= WRITE_ID_WRITE_SPICE_DATA;
           WRITE_ID_WRITE_SPICE_DATA: next <= WRITE_ID_HOLD_SPICE_DATA;
           WRITE_ID_HOLD_SPICE_DATA: begin
              if (write_id_ctr == 2'b00)
                next <= SET_NODE_ADDR;
              else
                next <= WRITE_ID_SET_SPICE_ADDR;
           end
           SET_NODE_ADDR: next <= READ_NODE_DATA;
           READ_NODE_DATA: next <= WRITE_NODE_SET_SPICE_ADDR;
           WRITE_NODE_SET_SPICE_ADDR: next <= WRITE_NODE_WRITE_SPICE_DATA;
```

```
            WRITE_NODE_WRITE_SPICE_DATA: next <= WRITE_NODE_HOLD_SPICE_DATA;
            WRITE_NODE_HOLD_SPICE_DATA: begin
                if (write_node_ctr != 2'b00) // do the next digit
                  next <= WRITE_NODE_SET_SPICE_ADDR;
                else begin
                   if (terminal_ctr == num_terminals) // move on
                     next <= WRITE_TYPE_SET_SPICE_ADDR;
                   else // do the next node
                     next <= SET_NODE_ADDR;
                end
            end
            WRITE_TYPE_SET_SPICE_ADDR: next <= WRITE_TYPE_WRITE_SPICE_DATA;
            WRITE_TYPE_WRITE_SPICE_DATA: next <= WRITE_TYPE_HOLD_SPICE_DATA;
            WRITE_TYPE_HOLD_SPICE_DATA: begin
                if (write_type_ctr == 2'b00)
                  next <= WRITE_VALUE_SET_SPICE_ADDR;
                else
                  next <= WRITE_TYPE_SET_SPICE_ADDR;
            end

            WRITE_VALUE_SET_SPICE_ADDR: next <= WRITE_VALUE_WRITE_SPICE_DATA;
            WRITE_VALUE_WRITE_SPICE_DATA: next <= WRITE_VALUE_HOLD_SPICE_DATA;
            WRITE_VALUE_HOLD_SPICE_DATA: begin
                if (write_value_ctr == 2'b00)
                   next <= WRITE_CR_SET_SPICE_ADDR;
                else
                  next <= WRITE_VALUE_SET_SPICE_ADDR;
            end
            WRITE_CR_SET_SPICE_ADDR: next <= WRITE_CR_WRITE_SPICE_DATA;
            WRITE_CR_WRITE_SPICE_DATA: next <= WRITE_CR_HOLD_SPICE_DATA;
            WRITE_CR_HOLD_SPICE_DATA: begin
                if (ideal_addr == 6'd63)
                  next <= WRITE_TERM_SET_SPICE_ADDR;
                else
                  next <= SET_IDEAL_ADDR;
            end

            WRITE_TERM_SET_SPICE_ADDR: next <= WRITE_TERM_WRITE_SPICE_DATA;
            WRITE_TERM_WRITE_SPICE_DATA: next <= WRITE_TERM_HOLD_SPICE_DATA;
            WRITE_TERM_HOLD_SPICE_DATA: next <= VRAM_CLEAR_INIT_ADDR;

            VRAM_CLEAR_INIT_ADDR: next <= VRAM_CLEAR_WRITE_DATA;
            VRAM_CLEAR_SET_ADDR: next <= VRAM_CLEAR_WRITE_DATA;
            VRAM_CLEAR_WRITE_DATA: next <= VRAM_CLEAR_HOLD_DATA;
            VRAM_CLEAR_HOLD_DATA: begin
                if (vram_write_addr == 16'd59999)
                  next <= VRAM_INIT_SPICE_ADDR;
                else
                  next <= VRAM_CLEAR_SET_ADDR;
            end

            VRAM_INIT_SPICE_ADDR: next <= VRAM_READ_SPICE_DATA;
            VRAM_SET_SPICE_ADDR: next <= VRAM_READ_SPICE_DATA;
            VRAM_READ_SPICE_DATA: next <= VRAM_SET_CHAR_ADDR;
            VRAM_SET_CHAR_ADDR: begin
                if (spice_data_read == CHR_EOF)
                  next <= TERMINATE;
                else if (spice_data_read == CHR_NEWLINE) // carriage return, write nothing but update line#
                  next <= VRAM_SET_VRAM_ADDR;
                else
                  next <= VRAM_READ_CHAR_DATA;
            end
            VRAM_READ_CHAR_DATA: next <= VRAM_SET_VRAM_ADDR;
            VRAM_SET_VRAM_ADDR: begin
                if (spice_data_read == CHR_NEWLINE)
                  next <= VRAM_SET_SPICE_ADDR;
                else
                  next <= VRAM_WRITE_VRAM_DATA;
            end
            VRAM_WRITE_VRAM_DATA: next <= VRAM_HOLD_VRAM_DATA;
            VRAM_HOLD_VRAM_DATA: begin
                if (vram_section_ctr != 2'd0)
                  next <= VRAM_SET_VRAM_ADDR;
                else if (bigchar_section_ctr != 5'd0)
                  next <= VRAM_SET_CHAR_ADDR;
                else
                  next <= VRAM_SET_SPICE_ADDR;
            end

            TERMINATE: next <= TERMINATE;
            default: next <= IDLE;
          endcase // case(state)
      end // else: !if(!active)
   end // always @ (state)
endmodule // minor_fsm_spice
```

```
module stack(clk, reset, stack_ctl, stack_data);
   input clk, reset;
   input [1:0] stack_ctl;

   inout [6:0] stack_data;

   // should we be driving the bus?
   reg         drive_bus;

   // value to hold result of pops
   reg [6:0]   bus_value;

   // tristate the bus if we are not currently driving it
   assign      stack_data = drive_bus ? bus_value : 7'bz;

   // stack_ctl parameters
   parameter   DEACTIVATE = 0;
   parameter   REST = 1;
   parameter   PUSH = 2;
   parameter   POP = 3;

   // special "start-of-stack" symbol
   parameter   SOS = 64;

   // RAM to use as backing
   reg [6:0]   stack_ram_input;
   wire [6:0]  stack_ram_output;
   reg [5:0]   stack_ram_addr;
   reg         stack_ram_we;
   // instantiate RAM
   stack_ram str(.clk(clk),
                 .addr(stack_ram_addr),
                 .din(stack_ram_input),
                 .dout(stack_ram_output),
                 .we(stack_ram_we));

   reg [3:0]   state, next;

   // state machine parameters
   parameter   IDLE = 0;
   parameter   INITIALIZE_1 = 1;
   parameter   INITIALIZE_2 = 2;
   parameter   INITIALIZE_3 = 3;
   parameter   WAIT_FOR_COMMAND = 4;
   parameter   PUSH_1 = 5;
   parameter   PUSH_2 = 6;
   parameter   PUSH_3 = 7;
   parameter   POP_1 = 8;
   parameter   POP_2 = 9;


   always @ (posedge clk or posedge reset) begin
      if (reset)
        state <= IDLE;
      else begin
         if (next == INITIALIZE_1)
           stack_ram_addr <= 7'd0;
         else if (next == PUSH_1)
           stack_ram_addr <= stack_ram_addr + 1;
         else if (next == POP_2) begin
            if (bus_value == SOS) // stack is empty, don't decrement
              stack_ram_addr <= stack_ram_addr;
            else
              stack_ram_addr <= stack_ram_addr + 6'd63; // subtract 1
         end

         if ((next == POP_1) || (next == POP_2))
           drive_bus <= 1'b1;
         else
           drive_bus <= 1'b0;

         if (next == POP_1)
           bus_value <= stack_ram_output;

         if ((next == INITIALIZE_2) || (next == PUSH_2))
           stack_ram_we <= 1'b1;
         else
           stack_ram_we <= 1'b0;

         if ((next == INITIALIZE_1) || (next == INITIALIZE_2) || (next == INITIALIZE_3))
           stack_ram_input <= SOS;
         else if (next == PUSH_1)
           stack_ram_input <= stack_data;
         else if ((next == PUSH_2) || (next == PUSH_3))
           stack_ram_input <= stack_ram_input;
         else
           stack_ram_input <= 7'bz;

         state <= next;
      end // else: !if(reset)
   end // always @ (posedge clk or posedge reset)

   always @ (state or stack_ctl) begin
      if (stack_ctl == DEACTIVATE)
        next <= IDLE;
      else begin
         case (state)
           IDLE: next <= INITIALIZE_1;
           INITIALIZE_1: next <= INITIALIZE_2;
           INITIALIZE_2: next <= INITIALIZE_3;
           INITIALIZE_3: next <= WAIT_FOR_COMMAND;
           WAIT_FOR_COMMAND: begin
              if (stack_ctl == PUSH)
                next <= PUSH_1;
              else if (stack_ctl == POP)
                next <= POP_1;
              else
                next <= WAIT_FOR_COMMAND;
           end
           PUSH_1: next <= PUSH_2;
           PUSH_2: next <= PUSH_3;
           PUSH_3: next <= WAIT_FOR_COMMAND;
           POP_1: next <= POP_2;
           POP_2: next <= WAIT_FOR_COMMAND;
         endcase // case(state)
      end // else: !if(stack_ctl == DEACTIVATE)
   end // always @ (state or stack_ctl)
endmodule // stack
```

```
// TODO: rename update_enq_list to check_enq_list
module nodefinder(clk, reset, active, node_data_out, ideal_data, node_data_in, node_addr, node_we, ideal_addr, finished
                  // debug
                  ,
                  state
                  // end debug
);
   input clk, reset, active;
   input [6:0] node_data_out;
   input [19:0] ideal_data;

   output [6:0] node_data_in;
   output [6:0] node_addr;
   output       node_we;
   output [5:0] ideal_addr;
   output       finished;
   output [5:0] state;

   reg [6:0]    node_data_in;
   reg [6:0]    node_addr;
   reg          node_we;
   reg [5:0]    ideal_addr;
   reg          finished;

   wire [2:0]   row_num;
   wire [2:0]   col_num;
   wire [7:0]   top_ptr;
   wire [7:0]   left_ptr;
   wire [7:0]   right_ptr;
   wire [7:0]   bot_ptr;

   assign       row_num = ideal_addr[5:3];
   assign       col_num = ideal_addr[2:0];

   assign       bot_ptr = (7 +
                           (15 * row_num) +
                           col_num);
   assign       top_ptr = bot_ptr - 15;
   assign       right_ptr = (15 * row_num) + col_num;
   assign       left_ptr = right_ptr - 1;

   // stack
   reg [1:0]    stack_ctl;
   wire [6:0]   stack_data;
   reg [5:0]    stack_data_input;
   reg          stack_data_drive;
   assign       stack_data = stack_data_drive ? {1'b0, stack_data_input} : 7'bz;

   // instantiate stack
   stack stk(.clk(clk),
             .reset(reset),
             .stack_ctl(stack_ctl),
             .stack_data(stack_data));

   // enqueued list (just a 64-bit register)
   reg [63:0]   enqueued_list;

   // array of registers to hold the node values for each edge
   // 0=top,1=left,2=right,3=bottom
   reg [6:0]    edge_node_value [0:3];

   // use these counters to track when to move on while waiting
   //   for pushes or pops to complete
   reg [1:0]    push_ctr;
   reg          pop_ctr;

   // counter tracks top,left,right,bottom node value population
   reg [1:0]    node_ctr;

   // counter to generate unique node assignments
   reg [6:0]    node_value_ctr;

   // counter to track scan of node table during conflict resolution
   //   and RAM initialization
   reg [6:0]    node_scan_ctr;

   // pointer to component type field for convenience
   wire [4:0]   block_type;
   assign       block_type = ideal_data[19:15];

   reg [5:0]    adjacent_block_addr;

   // array to track conflicting node assignments that we need to correct
   //   (for example, if we find a case where two nodes should be equal,
   //   but have different values, choose one value to be the winner, and
   //   scan the entire node assignment list, updating values equal to the
   //   loser to be the winner)
   // the first entry (0) is the winner, the rest are the losers
   reg [6:0]    conflict_node_values [0:3];
   reg [6:0]    conflict_ground_node;


   // random parameters
   parameter    BLANK_GRID_LOC = 5'd0;
   parameter    COMPONENT_THRESHOLD = 5'd11;
   parameter    CONNECTOR_THRESHOLD = 5'd21;
   parameter    NONEXISTENT_NODE = 7'd112;
   parameter    UNASSIGNED_NODE = 7'd113;
   parameter    GROUND_NODE = 5'd3;
   // special "start-of-stack" symbol
   parameter    SOS = 64;

   // stack_ctl parameters
   parameter   STACK_DEACTIVATE = 0;
   parameter   STACK_REST = 1;
   parameter   STACK_PUSH = 2;
   parameter   STACK_POP = 3;


   reg [5:0]    state, next;

   parameter    IDLE = 0;
   parameter    INITIALIZE = 1;
   parameter    INITIALIZE_RAM_SETUP_ADDR = 34;
   parameter    INITIALIZE_RAM_WRITE_DATA = 35;
   parameter    INITIALIZE_RAM_HOLD_DATA = 36;
   parameter    SET_INITIAL_IDEAL_ADDR = 2;
   parameter    INCREMENT_INITIAL_IDEAL_ADDR = 42;
   parameter    READ_INITIAL_IDEAL_DATA_1 = 43;
   parameter    READ_INITIAL_IDEAL_DATA_2 = 3;
   parameter    UPDATE_INITIAL_ENQ_LIST = 4;
   parameter    ANALYZE_CONNECTOR_SET_READ_NODE_ADDR = 5;
   parameter    ANALYZE_CONNECTOR_SETUP_READ_NODE_DATA = 6;
   parameter    ANALYZE_CONNECTOR_READ_NODE_DATA = 7;
   parameter    ANALYZE_CONNECTOR_CORRECT_NODE_DATA = 8;
   parameter    ANALYZE_CONNECTOR_SET_WRITE_NODE_ADDR = 9;
   parameter    ANALYZE_CONNECTOR_WRITE_NODE_DATA = 10;
   parameter    ANALYZE_CONNECTOR_HOLD_NODE_DATA = 11;
   parameter    ANALYZE_CONNECTOR_SET_SCAN_READ_NODE_ADDR = 12;
   parameter    ANALYZE_CONNECTOR_SETUP_SCAN_READ_NODE_DATA = 13;
   parameter    ANALYZE_CONNECTOR_SCAN_READ_NODE_DATA = 14;
```

```
   parameter    ANALYZE_CONNECTOR_SCAN_WRITE_NODE_DATA = 15;
   parameter    ANALYZE_CONNECTOR_SCAN_HOLD_NODE_DATA = 16;
   parameter    ANALYZE_COMPONENT_SET_READ_NODE_ADDR = 17;
   parameter    ANALYZE_COMPONENT_SETUP_READ_NODE_DATA = 18;
   parameter    ANALYZE_COMPONENT_READ_NODE_DATA = 19;
   parameter    ANALYZE_COMPONENT_SETUP_NODE_DATA = 20;
   parameter    ANALYZE_COMPONENT_WRITE_NODE_DATA = 21;
   parameter    ANALYZE_COMPONENT_HOLD_NODE_DATA = 22;
   parameter    ANALYZE_COMPONENT_SET_SCAN_READ_NODE_ADDR = 37;
   parameter    ANALYZE_COMPONENT_SETUP_SCAN_READ_NODE_DATA = 38;
   parameter    ANALYZE_COMPONENT_SCAN_READ_NODE_DATA = 39;
   parameter    ANALYZE_COMPONENT_SCAN_WRITE_NODE_DATA = 40;
   parameter    ANALYZE_COMPONENT_SCAN_HOLD_NODE_DATA = 41;
   parameter    UPDATE_ENQ_LIST = 23;
   parameter    PUSH_DO = 24;
   parameter    PUSH_WAIT = 25;
   parameter    POP_DO = 26;
   parameter    POP_READ = 27;
   parameter    POP_WAIT = 28;
   parameter    SET_IDEAL_ADDR = 29;
   parameter    READ_IDEAL_DATA = 30;
   parameter    DONE = 31;

   always @ (posedge clk or posedge reset) begin
      if (reset)
         state <= IDLE;
      else begin
         if (next == DONE)
           finished <= 1'b1;
         else
           finished <= 1'b0;

         // set the address for the ideal results ram
         if (next == SET_INITIAL_IDEAL_ADDR)
           ideal_addr <= 6'd0;
         else if (next == INCREMENT_INITIAL_IDEAL_ADDR)
           ideal_addr <= ideal_addr + 1;
//       else if (next == POP_READ) begin // setting next ideal address is actually done here, SET_IDEAL_ADDR is somewhat useless
         else if ((next == POP_WAIT) && (pop_ctr == 1'b0)) begin
            if (stack_data == SOS) // we terminate on this condition
              ideal_addr <= 6'bx;
            else
              ideal_addr <= stack_data[5:0];
         end
         else
           ideal_addr <= ideal_addr;


         // set the node address, set node data input, increment node scan counter
         if (next == INITIALIZE_RAM_SETUP_ADDR) begin
            node_addr <= node_scan_ctr;
            node_scan_ctr <= node_scan_ctr + 1;
            node_data_in <= UNASSIGNED_NODE;
         end
         else if ((next == INITIALIZE_RAM_WRITE_DATA) || (next == INITIALIZE_RAM_HOLD_DATA)) begin
            node_addr <= node_addr;
            node_data_in <= UNASSIGNED_NODE;
            node_scan_ctr <= node_scan_ctr;
         end
         else if ((next == ANALYZE_CONNECTOR_SET_READ_NODE_ADDR) || (next == ANALYZE_CONNECTOR_SET_WRITE_NODE_ADDR) || (next == ANALYZE_COMPONENT_SET_RE
AD_NODE_ADDR) || (next == UPDATE_ENQ_LIST)) begin
            // set node data input
            if (next == ANALYZE_CONNECTOR_SET_WRITE_NODE_ADDR)
              node_data_in <= edge_node_value[node_ctr];
            else
              node_data_in <= node_data_in;

            // set node address for  top, bottom, left, and right edges
            //   according to value of node_ctr
            // node_ctr is 0 for top, 1 for left, 2 for right, and 3 for bottom
            case (node_ctr)
              2'b00: begin // top
                 // grid locs 0..7 do not have a top edge
                 if (ideal_addr < 6'd8)
                   node_addr <= NONEXISTENT_NODE;
                 else // 7 + 15*((integer divide grid loc by 8) - 1) + location within row
                   node_addr <= top_ptr;
              end
              2'b01: begin // left
                 // grid locs evenly divisible by 8 do not have a left edge
                 if (col_num == 3'b000)
                   node_addr <= NONEXISTENT_NODE;
                 else // 15*(integer divide grid loc by 8) + (location within row - 1)
                   node_addr <= left_ptr;
              end
              2'b10: begin // right
                 // grid locs 1 short of being evenly divisible by 8 do not have a right edge
                 if (col_num == 3'b111)
                   node_addr <= NONEXISTENT_NODE;
                 else // 15*(integer divide grid loc by 8) + location within row
                   node_addr <= right_ptr;
              end
              2'b11: begin // bottom
                 // grid locs 56..63 do not have a bottom edge
                 if (ideal_addr > 6'd55)
                   node_addr <= NONEXISTENT_NODE;
                 else // 7 + 15*(integer divide grid loc by 8) + location within row
                   node_addr <= bot_ptr;
              end
            endcase // case(node_ctr)
            node_scan_ctr <= node_scan_ctr;
         end // if ((next == ANALYZE_CONNECTOR_SET_READ_NODE_ADDR) || (next == ANALYZE_CONNECTOR_SET_WRITE_NODE_ADDR) || (next == ANALYZE_COMPONENT_SET_
READ_NODE_ADDR))
         else if ((next == ANALYZE_CONNECTOR_SET_SCAN_READ_NODE_ADDR) || (next == ANALYZE_COMPONENT_SET_SCAN_READ_NODE_ADDR)) begin
            node_addr <= node_scan_ctr;
            // conflict_node_values[0] is the winner
            if (next == ANALYZE_COMPONENT_SET_SCAN_READ_NODE_ADDR)
              node_data_in <= 7'd0;
            else
              node_data_in <= conflict_node_values[0];
            node_scan_ctr <= node_scan_ctr + 1;
         end
         else if ((next == ANALYZE_CONNECTOR_SETUP_READ_NODE_DATA) || (next == ANALYZE_CONNECTOR_READ_NODE_DATA) || (next == ANALYZE_CONNECTOR_WRITE_NOD
E_DATA) || (next == ANALYZE_CONNECTOR_HOLD_NODE_DATA) || (next == ANALYZE_CONNECTOR_SETUP_SCAN_READ_NODE_DATA) || (next == ANALYZE_CONNECTOR_SCAN_READ_N
ODE_DATA) || (next == ANALYZE_CONNECTOR_SCAN_WRITE_NODE_DATA) || (next == ANALYZE_CONNECTOR_SCAN_HOLD_NODE_DATA) || (next == ANALYZE_COMPONENT_SETUP_REA
D_NODE_DATA) || (next == ANALYZE_COMPONENT_READ_NODE_DATA) || (next == ANALYZE_COMPONENT_WRITE_NODE_DATA) || (next == ANALYZE_COMPONENT_HOLD_NODE_DATA)
|| (next == ANALYZE_COMPONENT_SETUP_SCAN_READ_NODE_DATA) || (next == ANALYZE_COMPONENT_SCAN_READ_NODE_DATA) || (next == ANALYZE_COMPONENT_SCAN_WRITE_NOD
E_DATA) || (next == ANALYZE_COMPONENT_SCAN_HOLD_NODE_DATA)) begin
            node_addr <= node_addr;
            node_scan_ctr <= node_scan_ctr;
            node_data_in <= node_data_in;
         end
         else if (next == ANALYZE_COMPONENT_SETUP_NODE_DATA) begin
            // generate a new node assignment
            if (block_type == GROUND_NODE)
              node_data_in <= 7'd0;
            else
              node_data_in <= node_value_ctr;
            node_addr <= node_addr;
            node_scan_ctr <= node_scan_ctr;
```

```
         end
         else if ((next == SET_INITIAL_IDEAL_ADDR) || (next == SET_IDEAL_ADDR) || (next == INITIALIZE)) begin
            node_addr <= 7'bx;
            node_data_in <= 7'bx;
            // node_scan_ctr gets reset before initializing ram and any possible reassignment scan
            node_scan_ctr <= 7'b0;
         end
         else begin
            node_addr <= 7'bx;
            node_data_in <= 7'bx;
            node_scan_ctr <= node_scan_ctr;
         end

         // set write enable for node value ram
         if ((next == ANALYZE_CONNECTOR_WRITE_NODE_DATA) || (next == ANALYZE_CONNECTOR_SCAN_WRITE_NODE_DATA) || (next == ANALYZE_COMPONENT_WRITE_NODE_DA
TA) || (next == INITIALIZE_RAM_WRITE_DATA) || (next == ANALYZE_COMPONENT_SCAN_WRITE_NODE_DATA))
           node_we <= 1'b1;
         else
           node_we <= 1'b0;

         // read the node value
         // values updated in this block: top,left,right,bot edge node values,
         //   node value counter, conflict node values
         if ((next == ANALYZE_CONNECTOR_READ_NODE_DATA) || (next == ANALYZE_COMPONENT_READ_NODE_DATA)) begin
            case (node_ctr)
              2'b00: begin // top
                 // certain types do not have top edges
                 if ((block_type == 5'd0) || (block_type == 5'd1) || (block_type == 5'd5) || (block_type == 5'd7) || (block_type == 5'd9) || (block_type
 == 5'd21) || (block_type == 5'd23) || (block_type == 5'd25) || (block_type == 5'd29))
                   edge_node_value[0] <= NONEXISTENT_NODE;
                 else begin
                   if ((block_type == GROUND_NODE) && (node_data_out != 7'd0)) begin
                      conflict_ground_node <= node_data_out;
                      edge_node_value[0] <= 7'd0;
                   end
                   else begin
                      edge_node_value[0] <= node_data_out;
                      conflict_ground_node <= 7'd0;
                   end
                 end

                 edge_node_value[1] <= edge_node_value[1];
                 edge_node_value[2] <= edge_node_value[2];
                 edge_node_value[3] <= edge_node_value[3];
              end
              2'b01: begin // left
                 // certain types do not have left edges
                 if (! ((block_type == 5'd5) || (block_type == 5'd7) || (block_type == 5'd9) || (block_type == 5'd10) || (block_type == 5'd21) || (block
_type == 5'd25) || (block_type == 5'd26) || (block_type == 5'd28) || (block_type == 5'd29) || (block_type == 5'd30) || (block_type == 5'd31)))
                   edge_node_value[1] <= NONEXISTENT_NODE;
                 else
                   edge_node_value[1] <= node_data_out;
                 edge_node_value[0] <= edge_node_value[0];
                 edge_node_value[2] <= edge_node_value[2];
                 edge_node_value[3] <= edge_node_value[3];
              end
              2'b10: begin // right
                 // certain types do not have right edges
                 if (! ((block_type == 5'd5) || (block_type == 5'd7) || (block_type == 5'd9) || (block_type == 5'd11) || (block_type == 5'd21) || (block
_type == 5'd23) || (block_type == 5'd24) || (block_type == 5'd27) || (block_type == 5'd29) || (block_type == 5'd30) || (block_type == 5'd31)))
                   edge_node_value[2] <= NONEXISTENT_NODE;
                 else
                   edge_node_value[2] <= node_data_out;

                 edge_node_value[0] <= edge_node_value[0];
                 edge_node_value[1] <= edge_node_value[1];
                 edge_node_value[3] <= edge_node_value[3];
              end
              2'b11: begin // bottom
                 // certain types do not have bottom edges
                 if ((block_type == 5'd0) || (block_type == 5'd2) || (block_type == 5'd3) || (block_type == 5'd5) || (block_type == 5'd7) || (block_type
 == 5'd9) || (block_type == 5'd21) || (block_type == 5'd24) || (block_type == 5'd26) || (block_type == 5'd30))
                   edge_node_value[3] <= NONEXISTENT_NODE;
                 else
                   edge_node_value[3] <= node_data_out;

                 edge_node_value[0] <= edge_node_value[0];
                 edge_node_value[1] <= edge_node_value[1];
                 edge_node_value[2] <= edge_node_value[2];
              end
            endcase // case(node_ctr)
            node_value_ctr <= node_value_ctr;
         end // if (next == ANALYZE_CONNECTOR_READ_NODE_DATA)
         else if (next == ANALYZE_COMPONENT_SETUP_NODE_DATA) begin
            if (block_type == GROUND_NODE) begin
               edge_node_value[node_ctr] <= 7'd0;
               node_value_ctr <= node_value_ctr;
            end
            else begin
              edge_node_value[node_ctr] <= node_value_ctr;
               node_value_ctr <= node_value_ctr + 1;
            end
         end

         else if (next == ANALYZE_CONNECTOR_CORRECT_NODE_DATA) begin
            // if there are no valid edges with assigned values, generate a new value
            //   and assign it to all valid edges
            // NOTE: >= NONEXISTENT_NODE means edge is either nonexistent, or it exists but has no assigned value
            if ((edge_node_value[0] >= NONEXISTENT_NODE) &&
                (edge_node_value[1] >= NONEXISTENT_NODE) &&
                (edge_node_value[2] >= NONEXISTENT_NODE) &&
                (edge_node_value[3] >= NONEXISTENT_NODE)) begin
               if (edge_node_value[0] == UNASSIGNED_NODE)
                 edge_node_value[0] <= node_value_ctr;
               if (edge_node_value[1] == UNASSIGNED_NODE)
                 edge_node_value[1] <= node_value_ctr;
               if (edge_node_value[2] == UNASSIGNED_NODE)
                 edge_node_value[2] <= node_value_ctr;
               if (edge_node_value[3] == UNASSIGNED_NODE)
                 edge_node_value[3] <= node_value_ctr;
               // increment the node value counter for next assignment
               node_value_ctr <= node_value_ctr + 1;
               // no conflicts
            end
            // if there is just one edge with an assigned value, take that value
            //   and assign it to all edges
            else if (((edge_node_value[0] < NONEXISTENT_NODE) &&
                      (edge_node_value[1] >= NONEXISTENT_NODE) &&
                      (edge_node_value[2] >= NONEXISTENT_NODE) &&
                      (edge_node_value[3] >= NONEXISTENT_NODE)) ||
                     ((edge_node_value[0] >= NONEXISTENT_NODE) &&
                      (edge_node_value[1] < NONEXISTENT_NODE) &&
                      (edge_node_value[2] >= NONEXISTENT_NODE) &&
                      (edge_node_value[3] >= NONEXISTENT_NODE)) ||
                     ((edge_node_value[0] >= NONEXISTENT_NODE) &&
                      (edge_node_value[1] >= NONEXISTENT_NODE) &&
                      (edge_node_value[2] < NONEXISTENT_NODE) &&
                      (edge_node_value[3] >= NONEXISTENT_NODE)) ||
                     ((edge_node_value[0] >= NONEXISTENT_NODE) &&
                      (edge_node_value[1] >= NONEXISTENT_NODE) &&
```

```
                (edge_node_value[2] >= NONEXISTENT_NODE) &&
                (edge_node_value[3] < NONEXISTENT_NODE))) begin
            if (edge_node_value[0] < NONEXISTENT_NODE) begin
               if (edge_node_value[1] == UNASSIGNED_NODE)
                 edge_node_value[1] <= edge_node_value[0];
               if (edge_node_value[2] == UNASSIGNED_NODE)
                 edge_node_value[2] <= edge_node_value[0];
               if (edge_node_value[3] == UNASSIGNED_NODE)
                 edge_node_value[3] <= edge_node_value[0];
            end
            else if (edge_node_value[1] < NONEXISTENT_NODE) begin
               if (edge_node_value[0] == UNASSIGNED_NODE)
                 edge_node_value[0] <= edge_node_value[1];
               if (edge_node_value[2] == UNASSIGNED_NODE)
                 edge_node_value[2] <= edge_node_value[1];
               if (edge_node_value[3] == UNASSIGNED_NODE)
                 edge_node_value[3] <= edge_node_value[1];
            end
            else if (edge_node_value[2] < NONEXISTENT_NODE) begin
               if (edge_node_value[0] == UNASSIGNED_NODE)
                 edge_node_value[0] <= edge_node_value[2];
               if (edge_node_value[1] == UNASSIGNED_NODE)
                 edge_node_value[1] <= edge_node_value[2];
               if (edge_node_value[3] == UNASSIGNED_NODE)
                 edge_node_value[3] <= edge_node_value[2];
            end
            else if (edge_node_value[3] < NONEXISTENT_NODE) begin
               if (edge_node_value[0] == UNASSIGNED_NODE)
                 edge_node_value[0] <= edge_node_value[3];
               if (edge_node_value[1] == UNASSIGNED_NODE)
                 edge_node_value[1] <= edge_node_value[3];
               if (edge_node_value[2] == UNASSIGNED_NODE)
                 edge_node_value[2] <= edge_node_value[3];
            end
            else // something went seriously wrong
              $display("ERROR! Could not find the single unassigned block in ANALYZE_CONNECTOR_CORRECT_NODE_DATA");
            node_value_ctr <= node_value_ctr;
            // no conflicts
         end // if (((edge_node_value[0] < NONEXISTENT_NODE) &&...
         else begin
            // we have multiple assigned values.  it's possible that they
            //   are consistent, but it's too much work to write all the
            //   possible cases, so we'll just update all assignments.
            //   take the lowest value (to ultimately let 0 ground node take precedence
            //   without doing anything special) and assign it to all other nodes
            if ((edge_node_value[0] <= edge_node_value[1]) &&
                (edge_node_value[0] <= edge_node_value[2]) &&
                (edge_node_value[0] <= edge_node_value[3])) begin
               conflict_node_values[0] <= edge_node_value[0];
               // leave conflict_node_values UNASSIGNED_NODE if the "conflicting" value is actually a NONEXISTENT_NODE
               if (edge_node_value[1] != NONEXISTENT_NODE) begin
                  conflict_node_values[1] <= edge_node_value[1];
                  edge_node_value[1] <= edge_node_value[0];
               end
               if (edge_node_value[2] != NONEXISTENT_NODE) begin
                  conflict_node_values[2] <= edge_node_value[2];
                  edge_node_value[2] <= edge_node_value[0];
               end
               if (edge_node_value[3] != NONEXISTENT_NODE) begin
                  conflict_node_values[3] <= edge_node_value[3];
                  edge_node_value[3] <= edge_node_value[0];
               end
            end // if ((edge_node_value[0] <= edge_node_value[1]) &&...
            else if ((edge_node_value[1] <= edge_node_value[0]) &&
                     (edge_node_value[1] <= edge_node_value[2]) &&
                     (edge_node_value[1] <= edge_node_value[3])) begin
               conflict_node_values[0] <= edge_node_value[1];
               // leave conflict_node_value UNASSIGNED_NODE if the "conflicting" value is actually a NONEXISTENT_NODE
               if (edge_node_value[0] != NONEXISTENT_NODE) begin
                  conflict_node_values[1] <= edge_node_value[0];
                  edge_node_value[0] <= edge_node_value[1];
               end
               if (edge_node_value[2] != NONEXISTENT_NODE) begin
                  conflict_node_values[2] <= edge_node_value[2];
                  edge_node_value[2] <= edge_node_value[1];
               end
               if (edge_node_value[3] != NONEXISTENT_NODE) begin
                  conflict_node_values[3] <= edge_node_value[3];
                  edge_node_value[3] <= edge_node_value[1];
               end
            end // if ((edge_node_value[1] <= edge_node_value[0]) &&...
            else if ((edge_node_value[2] <= edge_node_value[0]) &&
                     (edge_node_value[2] <= edge_node_value[1]) &&
                     (edge_node_value[2] <= edge_node_value[3])) begin
               conflict_node_values[0] <= edge_node_value[2];
               // leave conflict_node_value UNASSIGNED_NODE if the "conflicting" value is actually a NONEXISTENT_NODE
               if (edge_node_value[0] != NONEXISTENT_NODE) begin
                  conflict_node_values[1] <= edge_node_value[0];
                  edge_node_value[0] <= edge_node_value[2];
               end
               if (edge_node_value[1] != NONEXISTENT_NODE) begin
                  conflict_node_values[2] <= edge_node_value[1];
                  edge_node_value[1] <= edge_node_value[2];
               end
               if (edge_node_value[3] != NONEXISTENT_NODE) begin
                  conflict_node_values[3] <= edge_node_value[3];
                  edge_node_value[3] <= edge_node_value[2];
               end
            end // if ((edge_node_value[2] <= edge_node_value[0]) &&...
            else if ((edge_node_value[3] <= edge_node_value[0]) &&
                     (edge_node_value[3] <= edge_node_value[1]) &&
                     (edge_node_value[3] <= edge_node_value[2])) begin
               conflict_node_values[0] <= edge_node_value[3];
               // leave conflict_node_value UNASSIGNED_NODE if the "conflicting" value is actually a NONEXISTENT_NODE
               if (edge_node_value[0] != NONEXISTENT_NODE) begin
                  conflict_node_values[1] <= edge_node_value[0];
                  edge_node_value[0] <= edge_node_value[3];
               end
               if (edge_node_value[1] != NONEXISTENT_NODE) begin
                  conflict_node_values[2] <= edge_node_value[1];
                  edge_node_value[1] <= edge_node_value[3];
               end
               if (edge_node_value[2] != NONEXISTENT_NODE) begin
                  conflict_node_values[3] <= edge_node_value[2];
                  edge_node_value[2] <= edge_node_value[3];
               end
            end // if ((edge_node_value[1] <= edge_node_value[0]) &&...
            else // something went seriously wrong
              $display("ERROR! Could not find a winner value among conflicts in ANALYZE_CONNECTOR_CORRECT_NODE_DATA");
            node_value_ctr <= node_value_ctr;
         end // else: !if(((edge_node_value[0] < NONEXISTENT_NODE) &&...
      end // if (next == ANALYZE_CONNECTOR_CORRECT_NODE_DATA)
      else if ((next == INITIALIZE) || (next == SET_INITIAL_IDEAL_ADDR) || (next == SET_IDEAL_ADDR)) begin
         if (next == INITIALIZE)
           // start at 1, node 0 is reserved for ground nodes
           node_value_ctr <= 7'd1;
         else
           node_value_ctr <= node_value_ctr;

         edge_node_value[0] <= UNASSIGNED_NODE;
```

```
            edge_node_value[1] <= UNASSIGNED_NODE;
            edge_node_value[2] <= UNASSIGNED_NODE;
            edge_node_value[3] <= UNASSIGNED_NODE;
            conflict_node_values[0] <= UNASSIGNED_NODE;
            conflict_node_values[1] <= UNASSIGNED_NODE;
            conflict_node_values[2] <= UNASSIGNED_NODE;
            conflict_node_values[3] <= UNASSIGNED_NODE;
         end
         else begin
            edge_node_value[0] <= edge_node_value[0];
            edge_node_value[1] <= edge_node_value[1];
            edge_node_value[2] <= edge_node_value[2];
            edge_node_value[3] <= edge_node_value[3];
            node_value_ctr <= node_value_ctr;
         end // else: !if((next == INITIALIZE) || (next == SET_INITIAL_IDEAL_ADDR) || (next == SET_IDEAL_ADDR))


         // increment node counter when appropriate
         if ((next == ANALYZE_CONNECTOR_READ_NODE_DATA) || (next == ANALYZE_CONNECTOR_HOLD_NODE_DATA) || (next == ANALYZE_COMPONENT_HOLD_NODE_DATA) || ( 
(next == PUSH_WAIT) && (push_ctr == 2'b01)))
           node_ctr <= node_ctr + 1;
         else if ((next == SET_INITIAL_IDEAL_ADDR) || (next == SET_IDEAL_ADDR))
           node_ctr <= 2'b00;
         else
           node_ctr <= node_ctr;

         if (next == UPDATE_ENQ_LIST)
           case (node_ctr)
             2'b00: adjacent_block_addr <= ideal_addr - 6'd8;
             2'b01: adjacent_block_addr <= ideal_addr - 6'd1;
             2'b10: adjacent_block_addr <= ideal_addr + 6'd1;
             2'b11: adjacent_block_addr <= ideal_addr + 6'd8;
           endcase // case(node_ctr)
         else
           adjacent_block_addr <= adjacent_block_addr;

         // increment the push counter
         if (next == PUSH_WAIT)
           push_ctr <= push_ctr + 1;
         else if (next == INITIALIZE)
           push_ctr <= 2'd0;
         else
           push_ctr <= push_ctr;

         // increment the pop counter
         if (next == POP_WAIT)
           pop_ctr <= pop_ctr + 1;
         else if (next == INITIALIZE)
           pop_ctr <= 1'b0;
         else
           pop_ctr <= pop_ctr;

         // during a push, update the enqueued list and activate the
         //   appropriate stack control signals
         if (next == PUSH_DO) begin
            stack_ctl <= STACK_PUSH;
            stack_data_input <= adjacent_block_addr;
            enqueued_list[adjacent_block_addr] <= 1'b1;
            stack_data_drive <= 1'b1;
         end
         // during a pop, activate the appropriate stack control signals
         else if (next == POP_DO) begin
            stack_ctl <= STACK_POP;
            stack_data_drive <= 1'b0;
            stack_data_input <= 6'bx;
         end
         // initialize stack and enqueued list when appropriate
         else if (next == INITIALIZE) begin
            stack_ctl <= STACK_DEACTIVATE;
            stack_data_drive <= 1'b0;
            stack_data_input <= 6'bx;
            enqueued_list <= 64'd0;
         end
         // otherwise, stack is at rest
         else if (next == UPDATE_INITIAL_ENQ_LIST) begin
            stack_ctl <= STACK_REST;
            stack_data_drive <= 1'b0;
            stack_data_input <= 6'bx;
            enqueued_list[ideal_addr] <= 1'b1;
         end
         else begin
            stack_ctl <= STACK_REST;
            stack_data_drive <= 1'b0;
            stack_data_input <= 6'bx;
         end

         state <= next;
      end
   end

   always @ (state or push_ctr or pop_ctr or active) begin
      if (~active)
        next <= IDLE;
      else begin
         case (state)
           IDLE: next <= INITIALIZE;
           INITIALIZE: next <= INITIALIZE_RAM_SETUP_ADDR;
           INITIALIZE_RAM_SETUP_ADDR: next <= INITIALIZE_RAM_WRITE_DATA;
           INITIALIZE_RAM_WRITE_DATA: next <= INITIALIZE_RAM_HOLD_DATA;
           INITIALIZE_RAM_HOLD_DATA: begin
              if (node_scan_ctr == 7'd112)
                next <= SET_INITIAL_IDEAL_ADDR;
              else
                next <= INITIALIZE_RAM_SETUP_ADDR;
           end
           SET_INITIAL_IDEAL_ADDR: next <= READ_INITIAL_IDEAL_DATA_1;
           INCREMENT_INITIAL_IDEAL_ADDR: next <= READ_INITIAL_IDEAL_DATA_1;
           READ_INITIAL_IDEAL_DATA_1: next <= READ_INITIAL_IDEAL_DATA_2;
           READ_INITIAL_IDEAL_DATA_2: begin
              // if the current component block is occupied, flag the
              //   appropriate entry in the enqueued list.  Otherwise,
              //   go back to set initial ideal address.
              if (block_type == BLANK_GRID_LOC)
                next <= INCREMENT_INITIAL_IDEAL_ADDR;
              else
                next <= UPDATE_INITIAL_ENQ_LIST;
           end
           UPDATE_INITIAL_ENQ_LIST: begin
              // if the grid loc holds a component, go to analyze component set read node address
              // otherwise, it holds a connector: go to analyze connector set read node address
              if (block_type >= CONNECTOR_THRESHOLD)
                next <= ANALYZE_CONNECTOR_SET_READ_NODE_ADDR;
              else if (block_type <= COMPONENT_THRESHOLD)
                next <= ANALYZE_COMPONENT_SET_READ_NODE_ADDR;
           end
           ANALYZE_CONNECTOR_SET_READ_NODE_ADDR: next <= ANALYZE_CONNECTOR_SETUP_READ_NODE_DATA;
           ANALYZE_CONNECTOR_SETUP_READ_NODE_DATA: next <= ANALYZE_CONNECTOR_READ_NODE_DATA;
           ANALYZE_CONNECTOR_READ_NODE_DATA: begin
              // if our counter has rolled over, then we've fully populated the node value registers,
              //   and can start checking for consistency
```

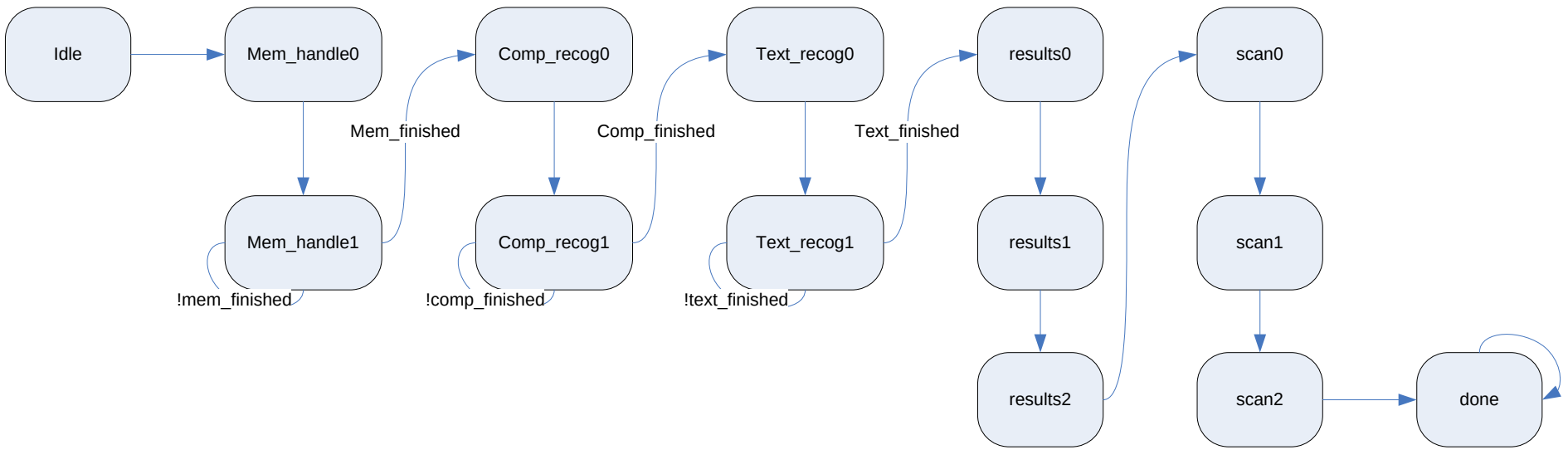
```
            // otherwise, go back to analyze connector set read node address
            if (node_ctr == 2'b00)
              next <= ANALYZE_CONNECTOR_CORRECT_NODE_DATA;
            else
              next <= ANALYZE_CONNECTOR_SET_READ_NODE_ADDR;
          end
          ANALYZE_CONNECTOR_CORRECT_NODE_DATA: next <= ANALYZE_CONNECTOR_SET_WRITE_NODE_ADDR;
          ANALYZE_CONNECTOR_SET_WRITE_NODE_ADDR: begin
            // we shouldn't write if there is no top,left, right, or bottom node
            //   for this grid location
            if (edge_node_value[node_ctr] >= NONEXISTENT_NODE)
              next <= ANALYZE_CONNECTOR_HOLD_NODE_DATA;
            else begin
               case (node_ctr)
                 2'b00: // top
                   if (ideal_addr < 6'd8)
                     next <= ANALYZE_CONNECTOR_HOLD_NODE_DATA;
                   else
                     next <= ANALYZE_CONNECTOR_WRITE_NODE_DATA;
                 2'b01: // left
                   if (ideal_addr[2:0] == 3'b000)
                     next <= ANALYZE_CONNECTOR_HOLD_NODE_DATA;
                   else
                     next <= ANALYZE_CONNECTOR_WRITE_NODE_DATA;
                 2'b10: // right
                   if (ideal_addr[2:0] == 3'b111)
                     next <= ANALYZE_CONNECTOR_HOLD_NODE_DATA;
                   else
                     next <= ANALYZE_CONNECTOR_WRITE_NODE_DATA;
                 2'b11: // bottom
                   if (ideal_addr > 6'd55)
                     next <= ANALYZE_CONNECTOR_HOLD_NODE_DATA;
                   else
                     next <= ANALYZE_CONNECTOR_WRITE_NODE_DATA;
               endcase // case(node_ctr)
            end // else: !if(edge_node_value[node_ctr] >= NONEXISTENT_NODE)
          end
          ANALYZE_CONNECTOR_WRITE_NODE_DATA: next <= ANALYZE_CONNECTOR_HOLD_NODE_DATA;
          ANALYZE_CONNECTOR_HOLD_NODE_DATA: begin
            // if the counter has not yet rolled over, then we need
            //   to loop back to set write node address
            // otherwise, move on.
            //   if there was a conflict, then we need to
            //     scan and update the node assignment table
            //   otherwise, update the enqueued list
            if (node_ctr != 2'b00)
              next <= ANALYZE_CONNECTOR_SET_WRITE_NODE_ADDR;
            else begin
               if ((conflict_node_values[0] == UNASSIGNED_NODE) &&
                   (conflict_node_values[1] == UNASSIGNED_NODE) &&
                   (conflict_node_values[2] == UNASSIGNED_NODE) &&
                   (conflict_node_values[3] == UNASSIGNED_NODE)) // no conflicts
                 next <= UPDATE_ENQ_LIST;
               else
                 next <= ANALYZE_CONNECTOR_SET_SCAN_READ_NODE_ADDR;
            end // else: !if(node_ctr != 2'b00)
          end // case: ANALYZE_CONNECTOR_HOLD_NODE_DATA
          ANALYZE_CONNECTOR_SET_SCAN_READ_NODE_ADDR: next <= ANALYZE_CONNECTOR_SETUP_SCAN_READ_NODE_DATA;
          ANALYZE_CONNECTOR_SETUP_SCAN_READ_NODE_DATA: next <= ANALYZE_CONNECTOR_SCAN_READ_NODE_DATA;
          ANALYZE_CONNECTOR_SCAN_READ_NODE_DATA: begin
            // if the assignment matches one of the "loser"
            //   values, update it by moving to scan_write
            // otherwise, go back to set_scan_read_node_addr
            // TODO: inefficiency: currently a conflict slot
            //   might actually hold the "winner" value
            //   (if there was a conflict among the 4 nodes,
            //    but two or three were the same),
            //   meaning we waste a few cycles needlessly
            //   rewriting a winner value.  fix if there's time
            if ((node_data_out != UNASSIGNED_NODE) &&
                ((node_data_out == conflict_node_values[1]) ||
                 (node_data_out == conflict_node_values[2]) ||
                 (node_data_out == conflict_node_values[3])))
              next <= ANALYZE_CONNECTOR_SCAN_WRITE_NODE_DATA;
            else begin
               // if the scan address counter has rolled over,
               //   move on to update the enqueued list
               // otherwise, loop back to set scan read node addr
//             if (node_scan_ctr == 7'd0)
               if (node_scan_ctr == 7'd112)
                 next <= UPDATE_ENQ_LIST;
               else
                 next <= ANALYZE_CONNECTOR_SET_SCAN_READ_NODE_ADDR;
            end // else: !if((node_data_out != UNASSIGNED_NODE) &&...
          end // case: ANALYZE_CONNECTOR_SCAN_READ_NODE_DATA
          ANALYZE_CONNECTOR_SCAN_WRITE_NODE_DATA: next <= ANALYZE_CONNECTOR_SCAN_HOLD_NODE_DATA;
          ANALYZE_CONNECTOR_SCAN_HOLD_NODE_DATA: begin
            // if the scan address counter has rolled over,
            //   move on to update the enqueued list
            // otherwise, loop back to set scan read node addr
//          if (node_scan_ctr == 7'd0)
            if (node_scan_ctr == 7'd112)
              next <= UPDATE_ENQ_LIST;
            else
              next <= ANALYZE_CONNECTOR_SET_SCAN_READ_NODE_ADDR;
          end

          ANALYZE_COMPONENT_SET_READ_NODE_ADDR: next <= ANALYZE_COMPONENT_SETUP_READ_NODE_DATA;
          ANALYZE_COMPONENT_SETUP_READ_NODE_DATA: next <= ANALYZE_COMPONENT_READ_NODE_DATA;
          ANALYZE_COMPONENT_READ_NODE_DATA: begin
            if (edge_node_value[node_ctr] == UNASSIGNED_NODE)
              next <= ANALYZE_COMPONENT_SETUP_NODE_DATA;
            else if ((block_type == GROUND_NODE) && (conflict_ground_node != 7'd0))
              next <= ANALYZE_COMPONENT_SET_SCAN_READ_NODE_ADDR;
            else
              next <= ANALYZE_COMPONENT_HOLD_NODE_DATA;
          end
          ANALYZE_COMPONENT_SETUP_NODE_DATA: next <= ANALYZE_COMPONENT_WRITE_NODE_DATA;
          ANALYZE_COMPONENT_WRITE_NODE_DATA: next <= ANALYZE_COMPONENT_HOLD_NODE_DATA;
          ANALYZE_COMPONENT_HOLD_NODE_DATA: begin
            // node ctr gets incremented for this state
            // if the node counter has rolled over, we've updated
            //   all pertinent nodes, so go on and update the
            //   enqueued list
            // otherwise, go back to set read node addr
            if (node_ctr == 2'b00)
              next <= UPDATE_ENQ_LIST;
            else
              next <= ANALYZE_COMPONENT_SET_READ_NODE_ADDR;
          end // case: ANALYZE_COMPONENT_HOLD_NODE_DATA
          ANALYZE_COMPONENT_SET_SCAN_READ_NODE_ADDR: next <= ANALYZE_COMPONENT_SETUP_SCAN_READ_NODE_DATA;
          ANALYZE_COMPONENT_SETUP_SCAN_READ_NODE_DATA: next <= ANALYZE_COMPONENT_SCAN_READ_NODE_DATA;
          ANALYZE_COMPONENT_SCAN_READ_NODE_DATA: begin
            if (node_data_out == conflict_ground_node)
              next <= ANALYZE_COMPONENT_SCAN_WRITE_NODE_DATA;
            else
              next <= ANALYZE_COMPONENT_SCAN_HOLD_NODE_DATA;
          end
          ANALYZE_COMPONENT_SCAN_WRITE_NODE_DATA: next <= ANALYZE_COMPONENT_SCAN_HOLD_NODE_DATA;
          ANALYZE_COMPONENT_SCAN_HOLD_NODE_DATA: begin
```

```
            if (node_scan_ctr == 7'd112)
              next <= ANALYZE_COMPONENT_HOLD_NODE_DATA;
            else
              next <= ANALYZE_COMPONENT_SET_SCAN_READ_NODE_ADDR;
          end

          UPDATE_ENQ_LIST: begin
            if ((node_addr != NONEXISTENT_NODE) && (edge_node_value[node_ctr] < NONEXISTENT_NODE) && (enqueued_list[adjacent_block_addr] == 1'b0))
              // push it on the stack
              next <= PUSH_DO;
            else
              // skip it
              next <= PUSH_WAIT;
          end
          PUSH_DO: next <= PUSH_WAIT;
          PUSH_WAIT: begin
            if (push_ctr == 2'b00) begin
              if (node_ctr == 2'b00)
                next <= POP_DO;
              else
                next <= UPDATE_ENQ_LIST;
            end
            else
              next <= PUSH_WAIT;
          end
          POP_DO: next <= POP_READ;
          POP_READ: next <= POP_WAIT; // reading is actually done during first POP_WAIT
          POP_WAIT: begin
            if (pop_ctr == 1'b0)
              next <= SET_IDEAL_ADDR;
            else begin
              if (stack_data == SOS)
                next <= DONE;
              else
                next <= POP_WAIT;
            end
          end
          SET_IDEAL_ADDR: next <= READ_IDEAL_DATA;
          READ_IDEAL_DATA: begin
            if (block_type >= CONNECTOR_THRESHOLD)
              next <= ANALYZE_CONNECTOR_SET_READ_NODE_ADDR;
            else if (block_type <= COMPONENT_THRESHOLD)
              next <= ANALYZE_COMPONENT_SET_READ_NODE_ADDR;
          end
          DONE: next <= DONE;
          default: next <= IDLE;
        endcase // case(state)
      end // else: !if(~active)
    end // always @ (state or push_ctr or pop_ctr or active)
endmodule // nodefinder
```

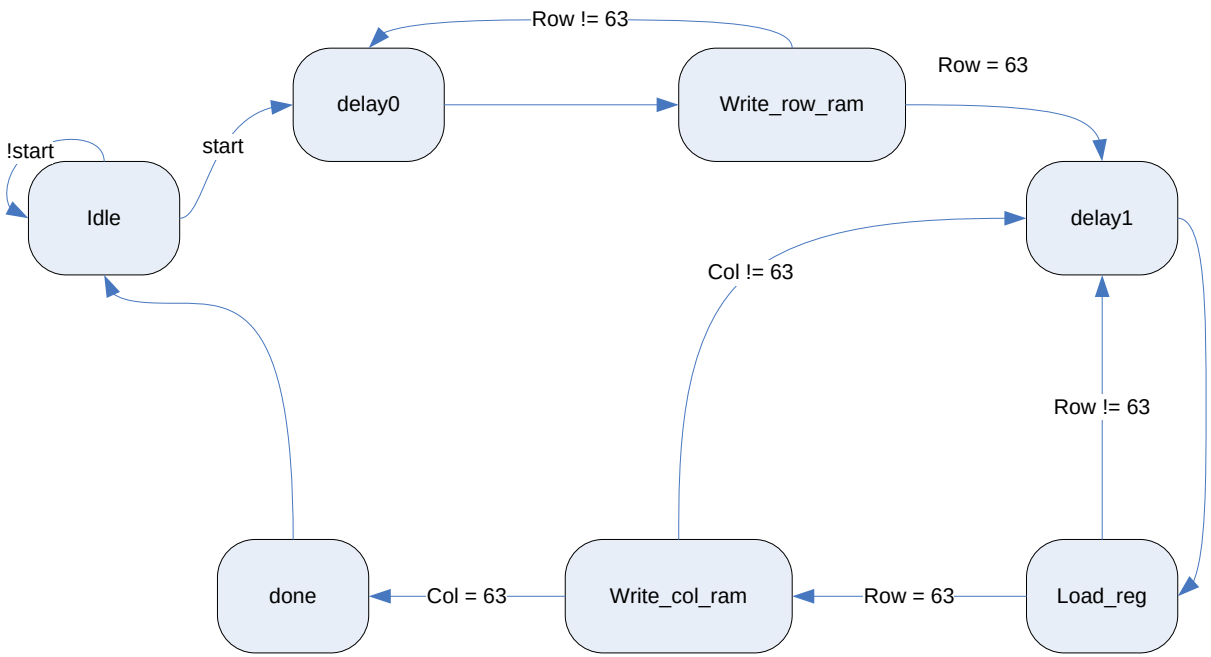
```
module serial_fsm(clk,reset_sync, start, finished,
                  ascii_out,
                          char_add, char_out,
                          TxD, TxD_busy,
                          //debug
                          transmitter_start,
                          state,
                          fsm_start


                          //end debug
                          );

    input clk, reset_sync, start;
    output finished;
    reg finished;


    input [7:0] ascii_out;
    input [4:0] char_out;

    output [9:0] char_add;
    reg [9:0] char_add;

    output TxD;
    output TxD_busy;
    output transmitter_start, fsm_start;
    reg transmitter_start;
    wire transmitter_finished;
    reg start_flag, fsm_start;


    output [5:0] state;
    reg [5:0] state;
    parameter idle = 0;
    parameter read_char0 = 1;
    parameter read_char1 = 2;
    parameter read_ascii0 = 7;
    parameter read_ascii1 = 5;
    parameter transmit_char0 = 3;
    parameter transmit_char1 = 4;
    parameter done = 6;



    async_transmitter my_async_transmitter(.clk(clk), .start(transmitter_start),
                                           .TxD_data(ascii_out), .TxD(TxD),
                                                                 .TxD_busy(TxD_busy), .finished(transmitter_finished));

^M
always @ (posedge clk) begin^M
if (start && !start_flag) begin
        start_flag <= 1;
        fsm_start <= 1;
        end
else if (start && start_flag) begin
        fsm_start <= 0;
        end
else    begin
        start_flag <= 0;
        fsm_start <= 0;
end
^M
case (state)^M
idle:
        if (fsm_start) state <= read_char0;
                       else begin
                           state <= idle;
                           transmitter_start <= 0;
                           char_add <= 0;
                       end

    read_char0: state <= read_char1;

    read_char1: if (char_out == 5'd30) state <= done;
    else state <= read_ascii0;

    read_ascii0: state <= read_ascii1;
    read_ascii1: begin
    state <= transmit_char0;
    transmitter_start <= 1;
    end

    transmit_char0: begin
            transmitter_start <= 0;
            state <= transmit_char1;
    end

    transmit_char1:
    if (transmitter_finished)
                        if (&(char_add)) state <= done;
                        else begin
                    char_add <= char_add + 1;
                    state <= read_char0;
                        end
    else state <= transmit_char1;


    done: begin
    state <= idle;
    end

    endcase
    if (reset_sync) begin
    state <= idle;
    transmitter_start <= 0;
    char_add <= 0;
    fsm_start <= 0;
    end
    end



endmodule
```

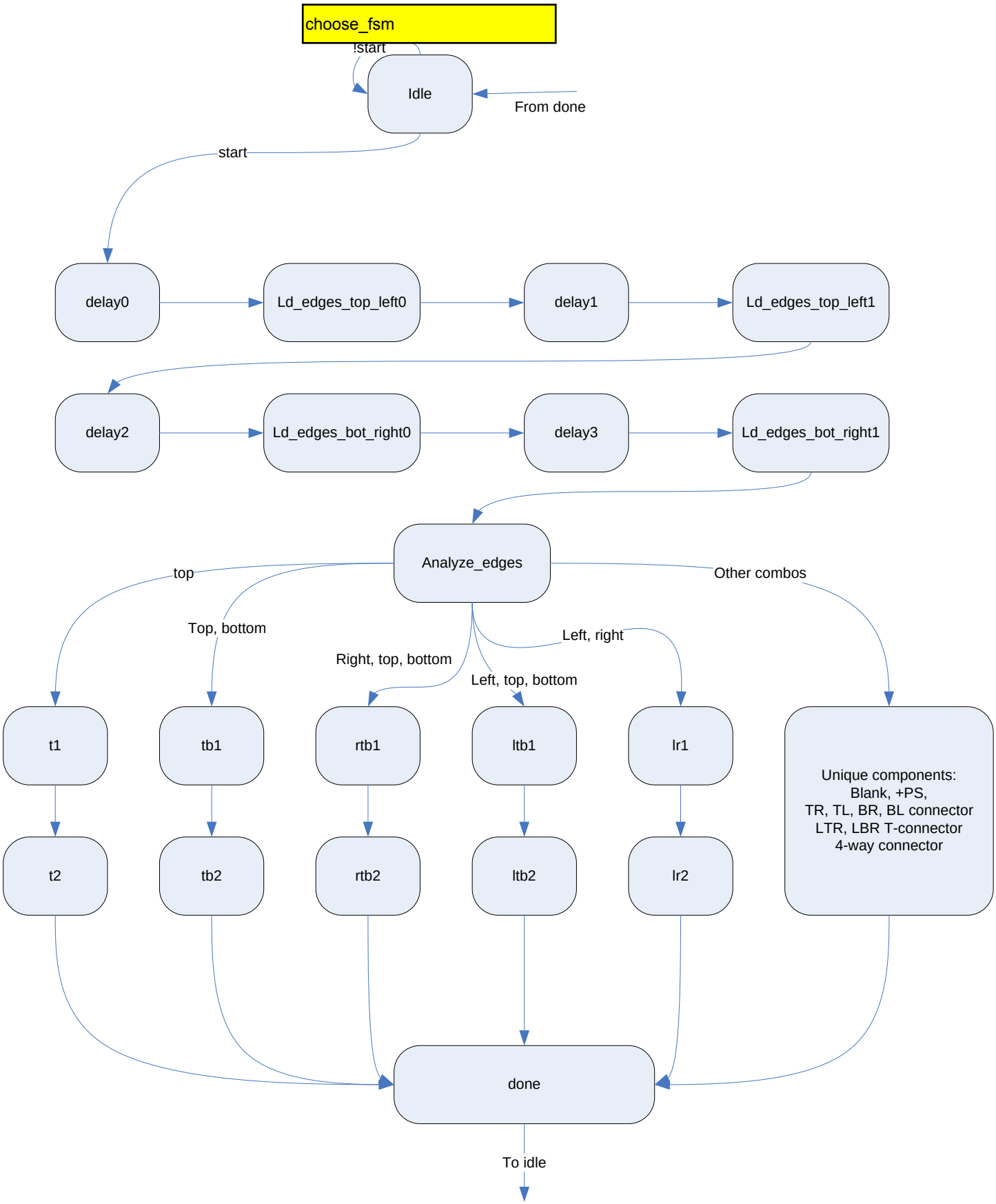
```
// taken from http://www.fpga4fun.com

module async_transmitter(clk, start, TxD_data, TxD, TxD_busy, finished);
input clk, start;
input [7:0] TxD_data;
output TxD, TxD_busy;
output finished;
reg finished;


parameter ClkFrequency = 50142857; // 50.1MHz
parameter Baud = 115200;

// Baud generator
parameter BaudGeneratorAccWidth = 16;
parameter BaudGeneratorInc = ((Baud<<(BaudGeneratorAccWidth-4))+(ClkFrequency>>5))/(ClkFrequency>>4);
reg [BaudGeneratorAccWidth:0] BaudGeneratorAcc;
wire BaudTick = BaudGeneratorAcc[BaudGeneratorAccWidth];
wire TxD_busy;
always @(posedge clk) if(TxD_busy) BaudGeneratorAcc <= BaudGeneratorAcc[BaudGeneratorAccWidth-1:0] + BaudGeneratorInc;

// Transmitter state machine
reg [3:0] state;
assign TxD_busy = (state!=0);

always @(posedge clk)
case(state)
  4'b0000: begin
           finished <= 0;
           if(start) state <= 4'b0100;
                 end
  4'b0100: if(BaudTick) state <= 4'b1000;  // start
  4'b1000: if(BaudTick) state <= 4'b1001;  // bit 0
  4'b1001: if(BaudTick) state <= 4'b1010;  // bit 1
  4'b1010: if(BaudTick) state <= 4'b1011;  // bit 2
  4'b1011: if(BaudTick) state <= 4'b1100;  // bit 3
  4'b1100: if(BaudTick) state <= 4'b1101;  // bit 4
  4'b1101: if(BaudTick) state <= 4'b1110;  // bit 5
  4'b1110: if(BaudTick) state <= 4'b1111;  // bit 6
  4'b1111: if(BaudTick) state <= 4'b0001;  // bit 7
  4'b0001: if(BaudTick) state <= 4'b0010;  // stop1
  4'b0010: if(BaudTick) begin
           state <= 4'b0000;
                 finished <= 1;
                 end  // stop2
  default: if(BaudTick) state <= 4'b0000;
endcase

// Output mux
reg muxbit;
always @(state[2:0] or TxD_data)
case(state[2:0])
  0: muxbit <= TxD_data[0];
  1: muxbit <= TxD_data[1];
  2: muxbit <= TxD_data[2];
  3: muxbit <= TxD_data[3];
  4: muxbit <= TxD_data[4];
  5: muxbit <= TxD_data[5];
  6: muxbit <= TxD_data[6];
  7: muxbit <= TxD_data[7];
endcase

// Put together the start, data and stop bits
reg TxD;
always @(posedge clk) TxD <= (state<4) | (state[3] & muxbit);  // register the output to make it glitch free

endmodule
```

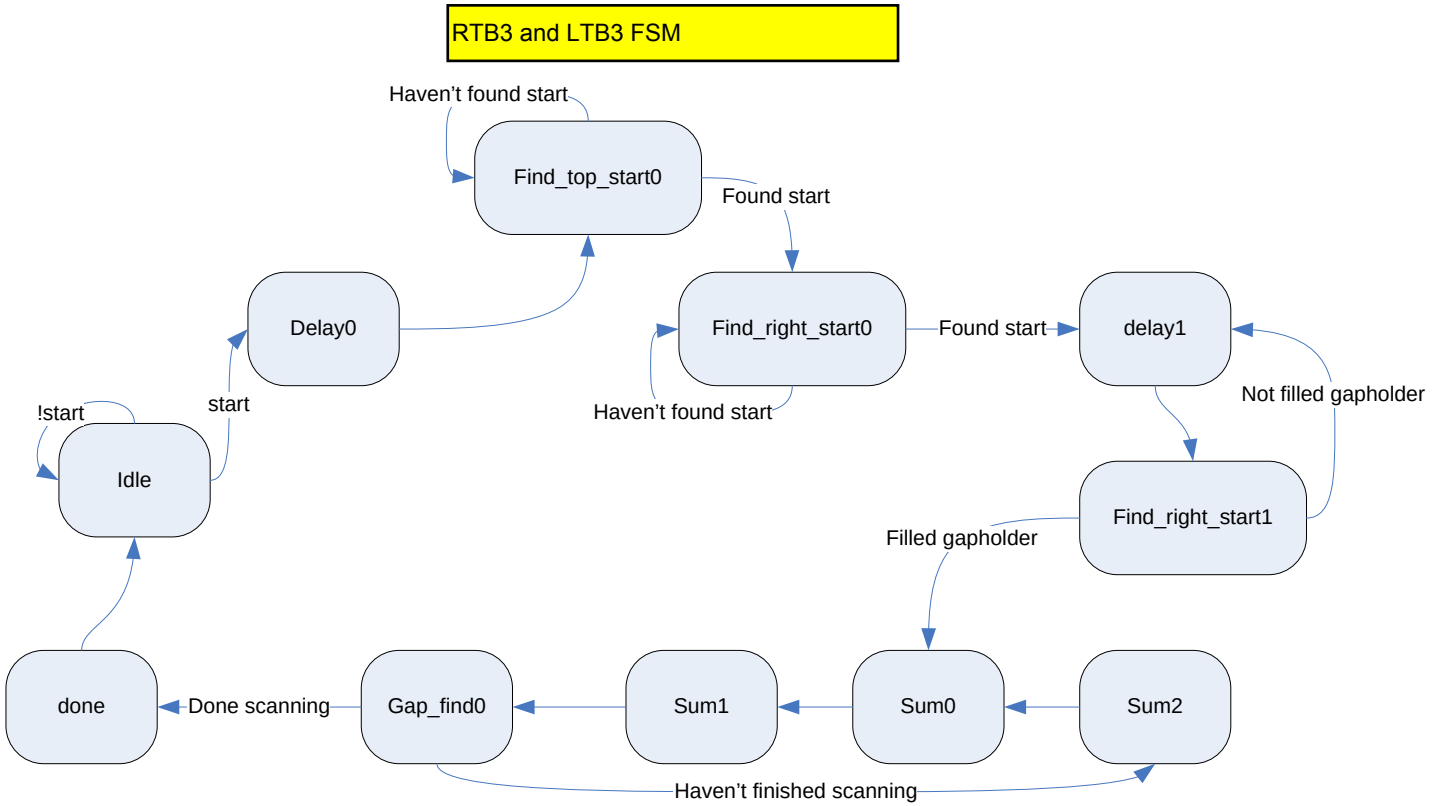
major fsm


Idle
Mem_handle0
Mem_handle1
!mem_finished
Mem_finished
Comp_recog0
Comp_recog1
!comp_finished
Comp_finished
Text_recog0
Text_recog1
!text_finished
Text_finished
results0
results1
results2
scan0
scan1
scan2
done

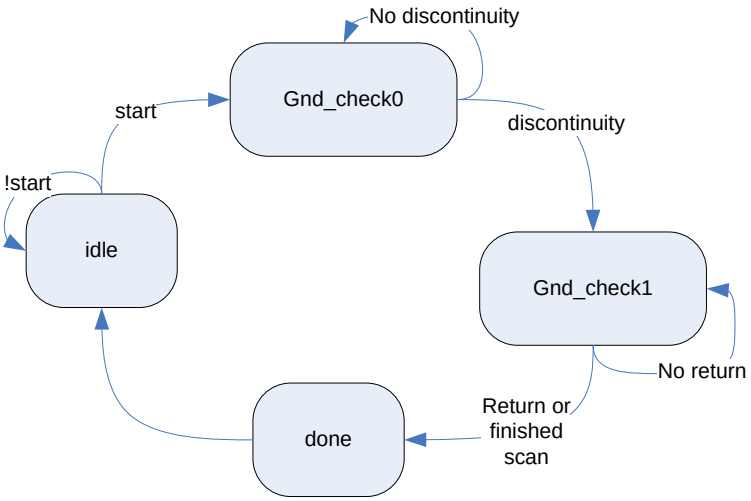
mem_fsm

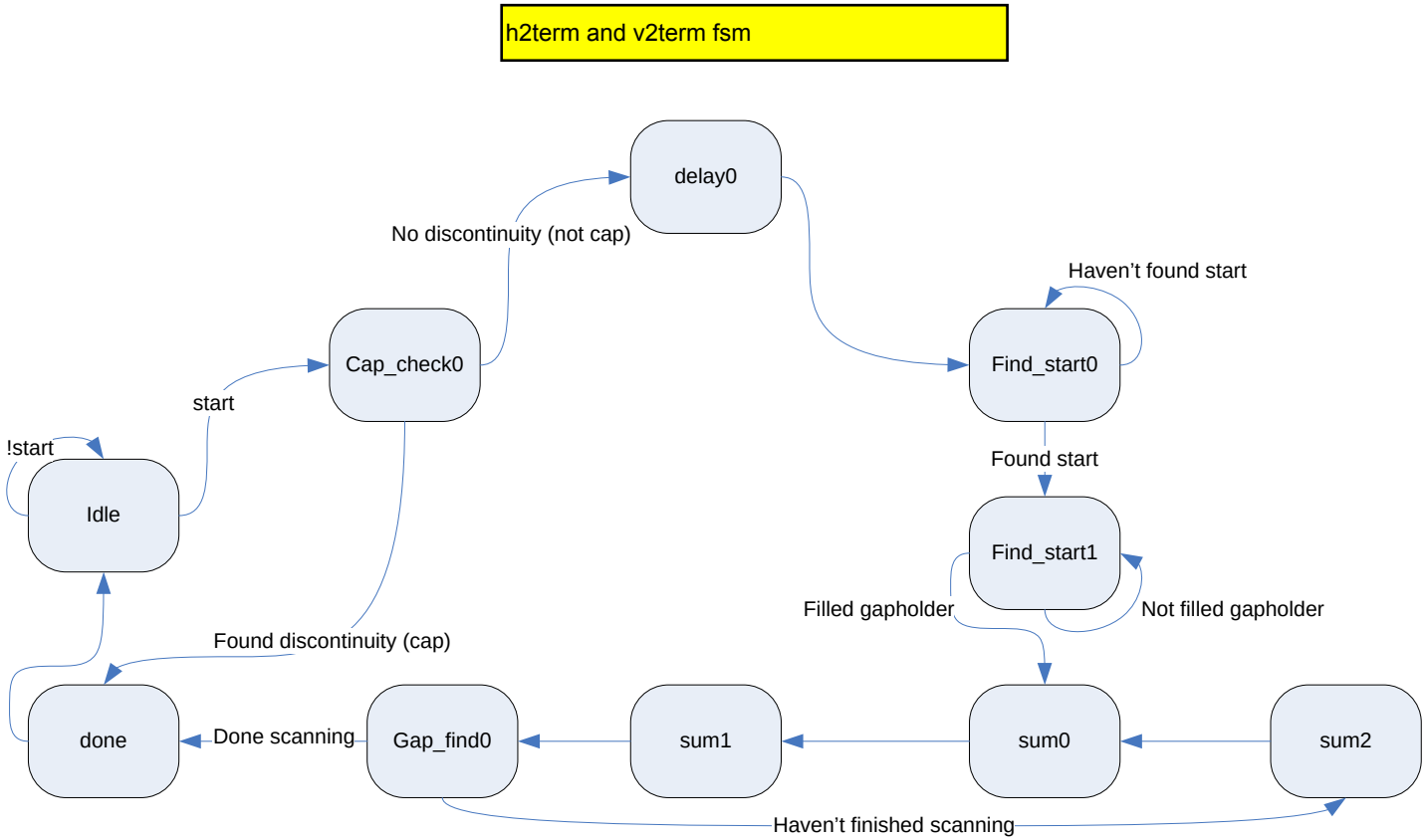

Row != 63
delay0
Write_row_ram
Row = 63
!start
start
Idle
delay1
Col != 63
Row != 63
done
Col = 63
Write_col_ram
Row = 63
Load_reg

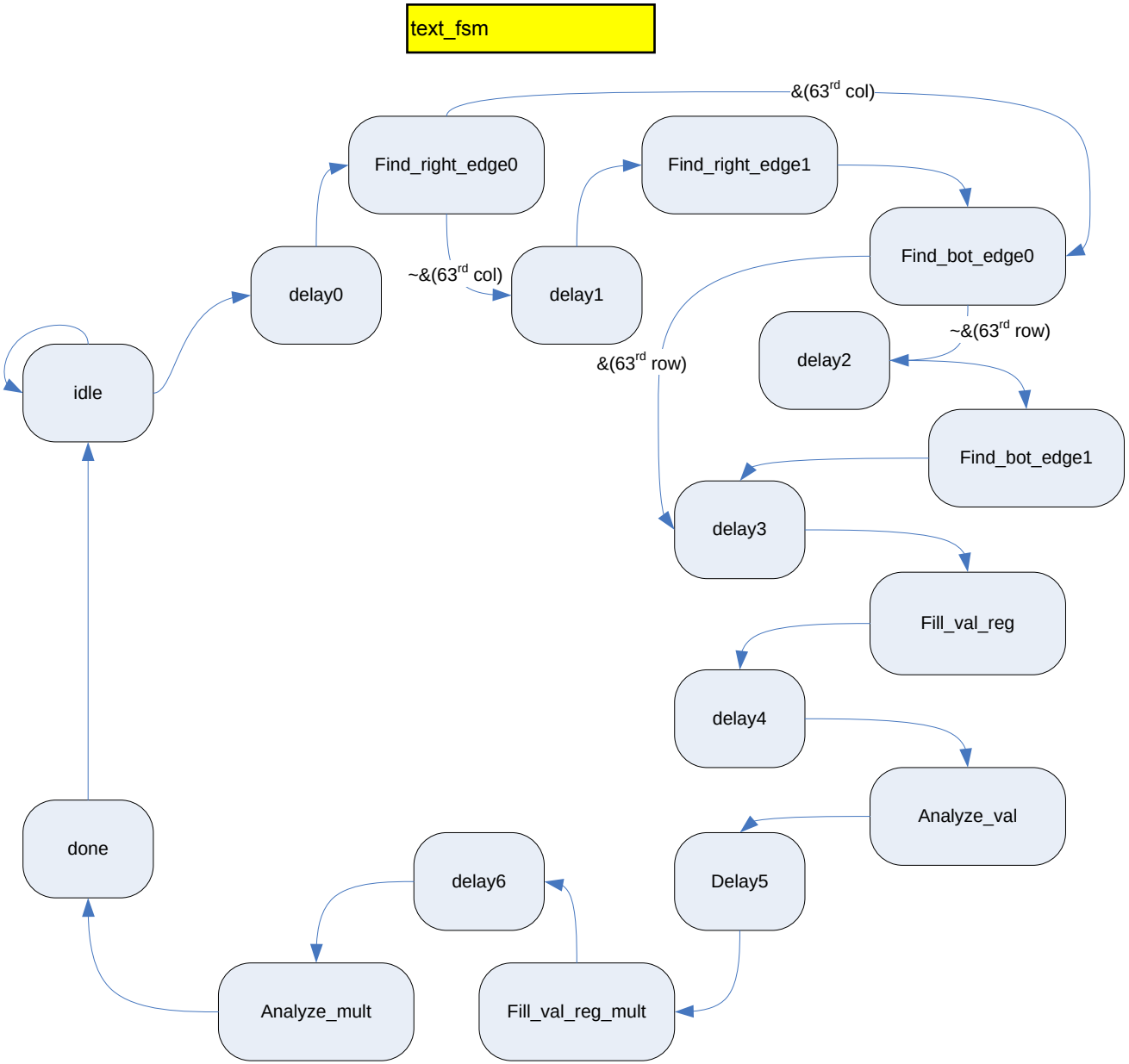

choose_fsm
!start
Idle
From done
start
delay0
Ld_edges_top_left0
delay1
Ld_edges_top_left1
delay2
Ld_edges_bot_right0
delay3
Ld_edges_bot_right1
Analyze_edges
top
Top, bottom
Right, top, bottom
Left, top, bottom
Left, right
Other combos
t1
tb1
rtb1
ltb1
lr1
t2
tb2
rtb2
ltb2
lr2
Unique components:
Blank, +PS,
TR, TL, BR, BL connector
LTR, LBR T-connector
4-way connector
done
To idle

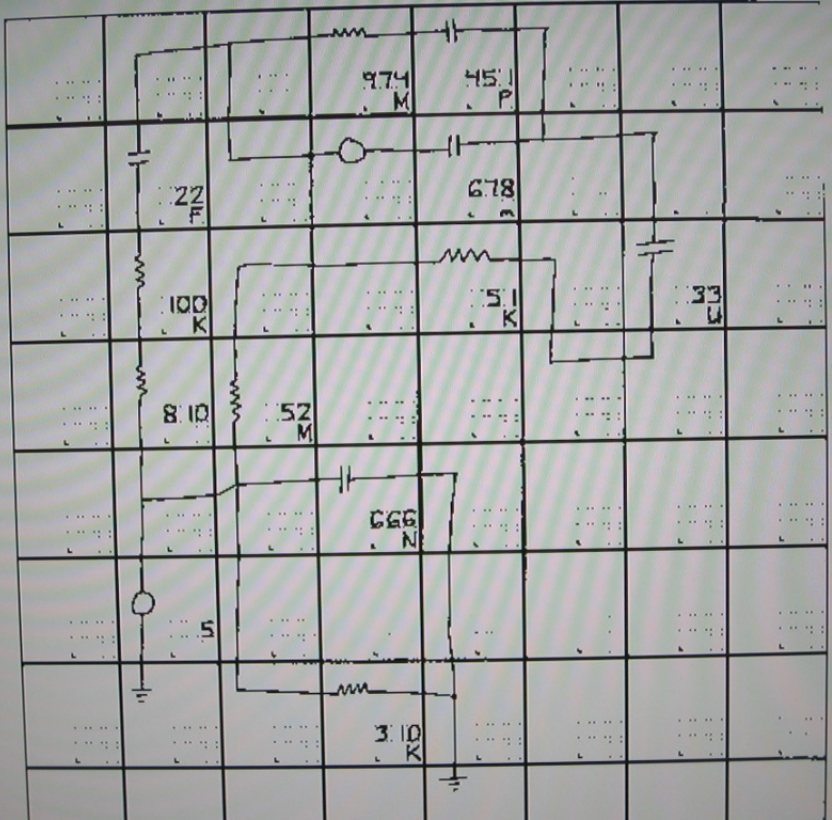

RTB3 and LTB3 FSM
Haven’t found start
Find_top_start0
Found start
Delay0
Find_right_start0
Found start
delay1
Not filled gapholder
!start
start
Haven’t found start
Idle
Find_right_start1
Filled gapholder
done
Done scanning
Gap_find0
Sum1
Sum0
Sum2
Haven’t finished scanning

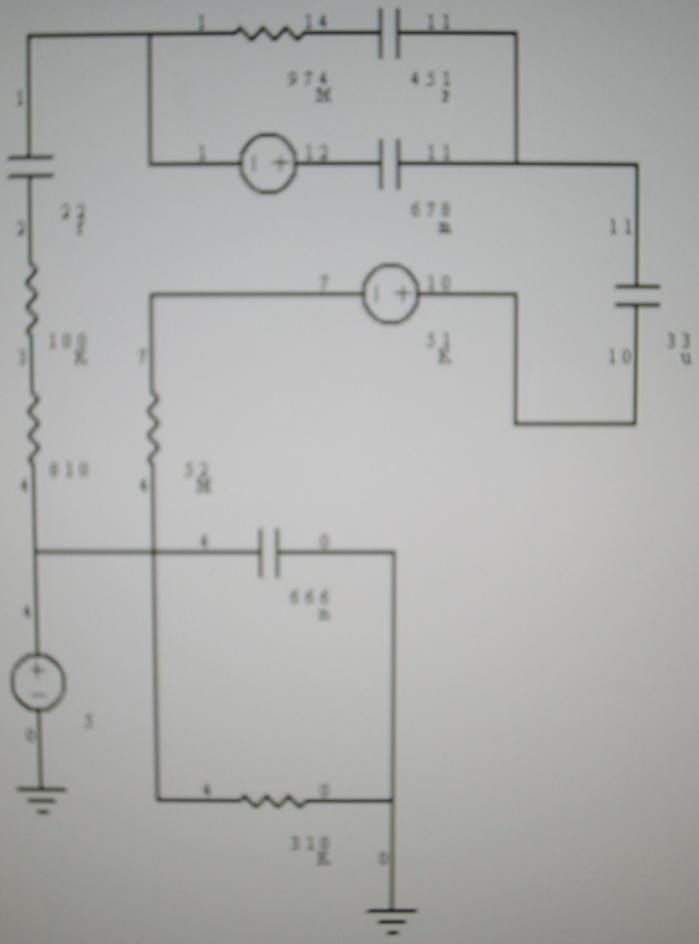
t1term fsm


No discontinuity
Gnd_check0
start
discontinuity
!start
idle
Gnd_check1
No return
Return or
finished
scan
done

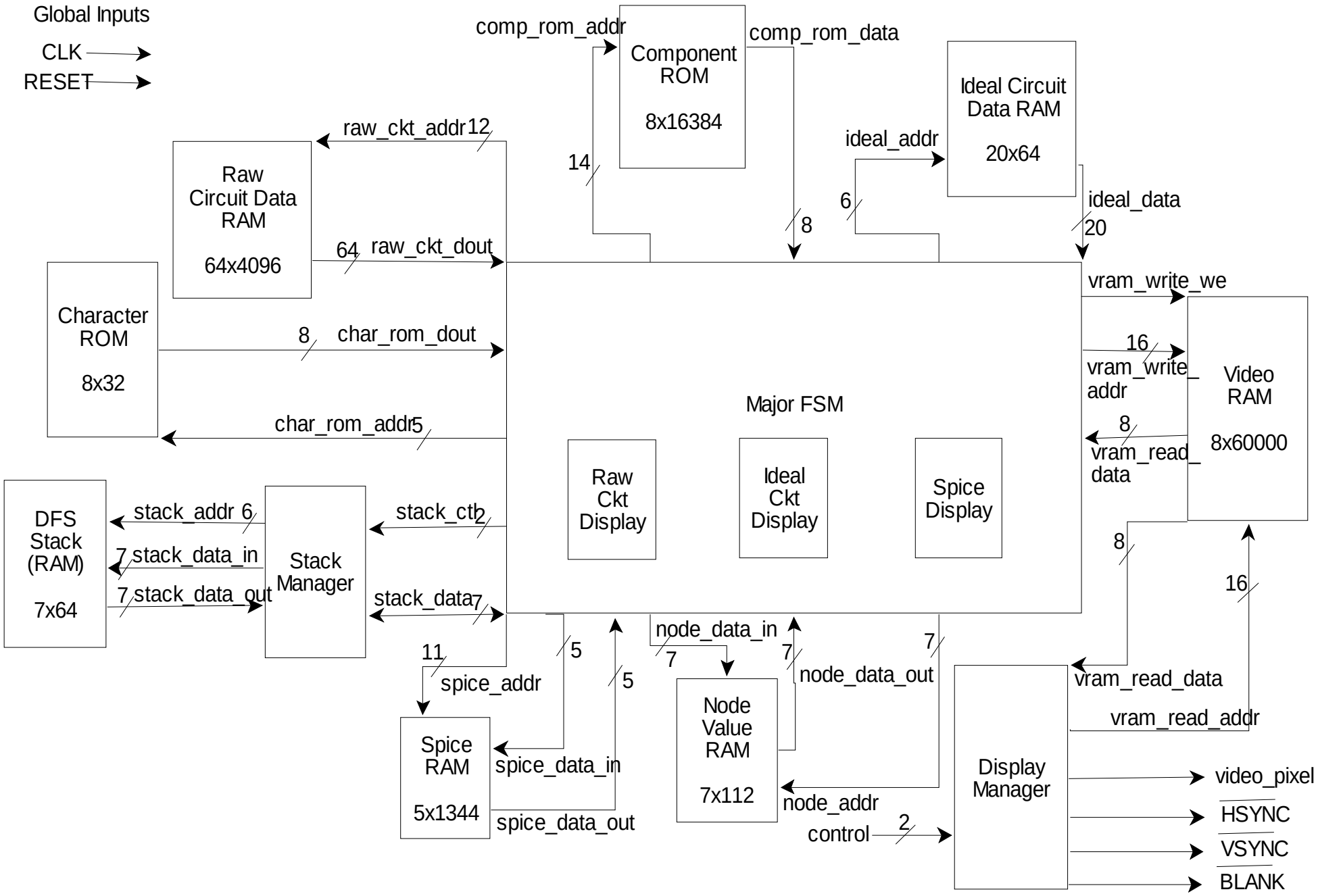

h2term and v2term fsm
delay0
No discontinuity (not cap)
Haven’t found start
Cap_check0
Find_start0
start
!start
Found start
Idle
Find_start1
Filled gapholder
Not filled gapholder
Found discontinuity (cap)
done
Done scanning
Gap_find0
sum1
sum0
sum2
Haven’t finished scanning


text_fsm
&(63rd col)
Find_right_edge0
Find_right_edge1
Find_bot_edge0
delay0
~&(63rd col)
delay1
~&(63rd row)
delay2
&(63rd row)
idle
Find_bot_edge1
delay3
Fill_val_reg
delay4
Analyze_val
done
delay6
Delay5
Analyze_mult
Fill_val_reg_mult

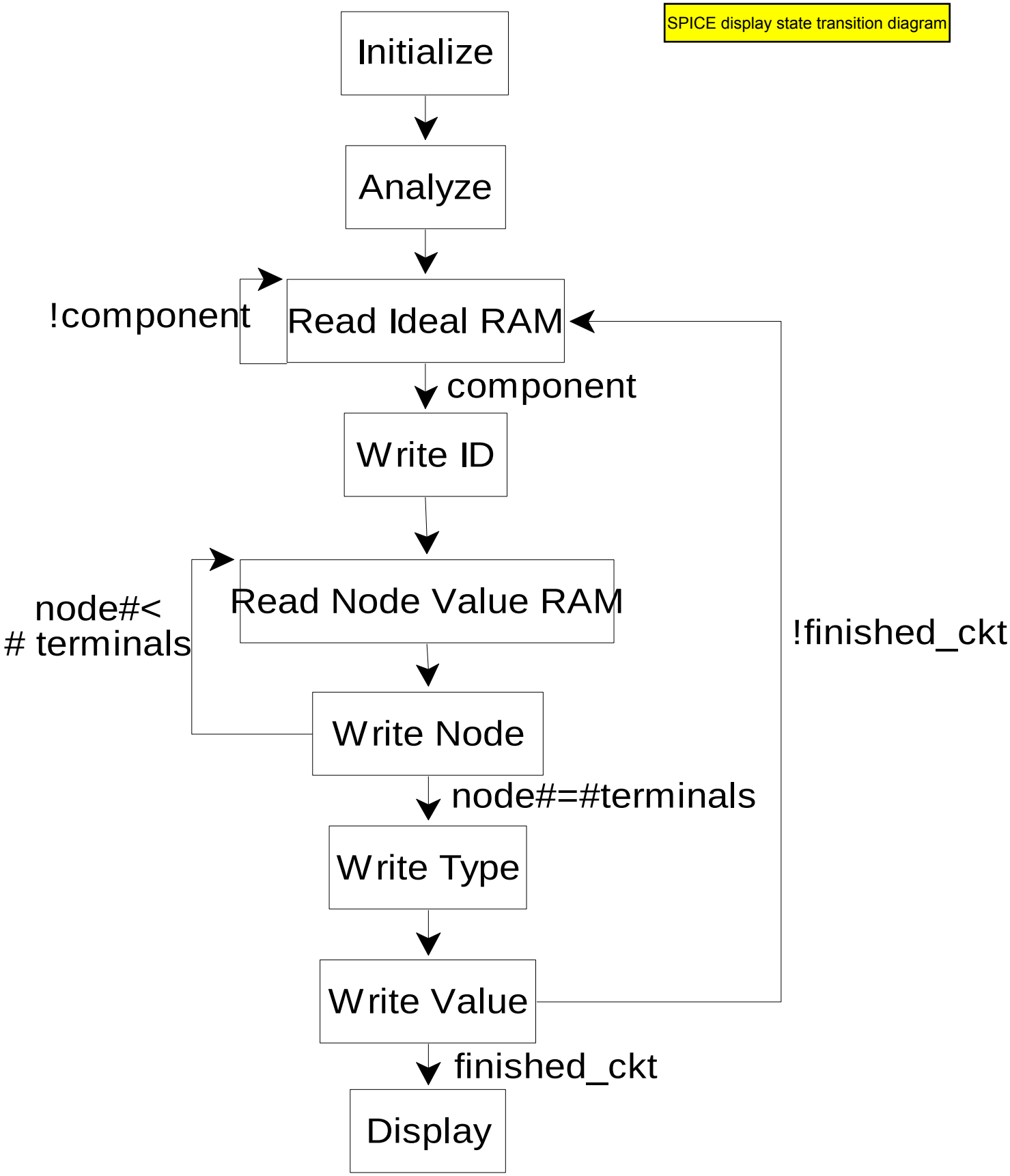

974
M
451
P
22
F
678
100
K
5.1
K
33
8.10
52
M
666
N
5
3.10
K


1
14
11
974
M
451
P
1
1
12
11
678
m
11
2
22
f
7
10
3
100
K
7
51
K
10
33
u
4
810
4
52
M
4
0
666
m
4
3
0
4
0
310
K
0

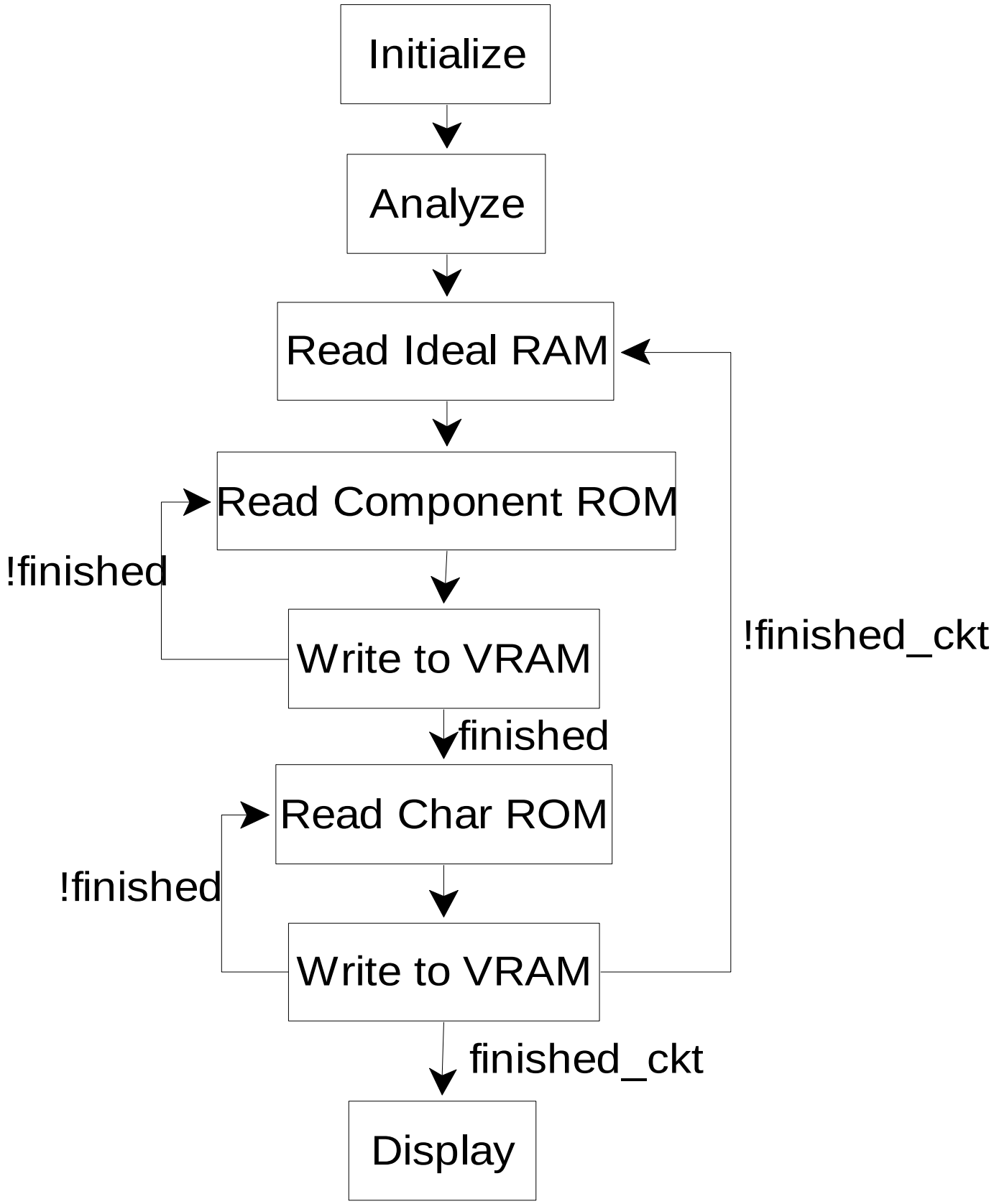
R00	14	1		974M
C00	11	14		451p
C01	1	2		22f
V00	12	1	DC	
C02	11	12		678m
R01	2	3		100K
V01	10	7	DC	51K
C03	11	10		33u
R02	3	4		810
R03	7	4		52M
C04	0	4		666n
V02	4	0	DC	5
R04	0	4		310K

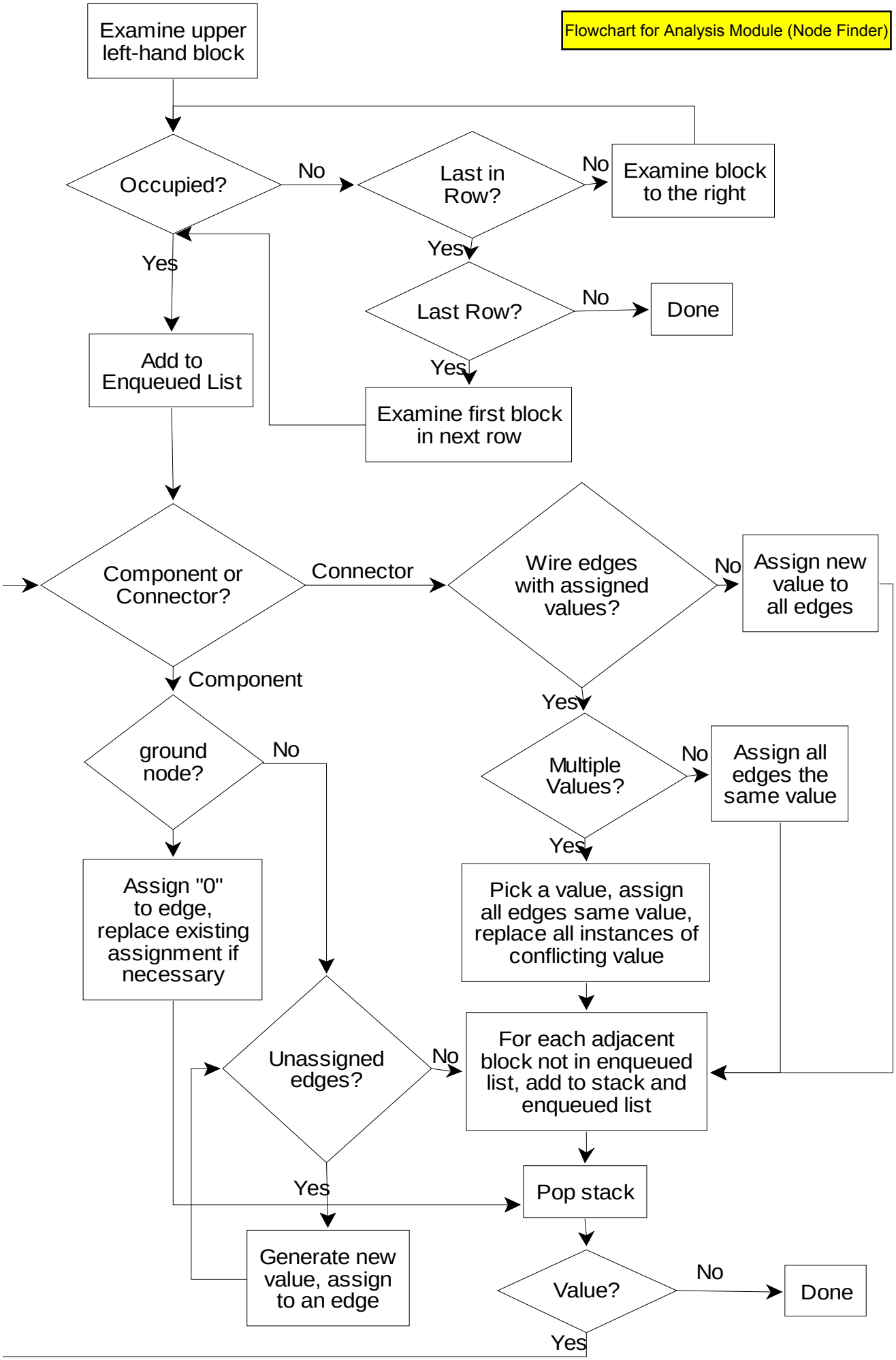


Block Diagram for Analysis and Video modules


SPICE display state transition diagram
Initialize
Analyze
!component
Read Ideal RAM
component
Write ID
Read Node Value RAM
node#<
terminals
Write Node
node#=#terminals
Write Type
Write Value
!finished_ckt
finished_ckt
Display

Ideal Circuit Display State Transition Diagram




Flowchart for Analysis Module (Node Finder)
Examine upper left-hand block
Occupied?
No
Last in Row?
No
Examine block to the right
Yes
Yes
Last Row?
No
Done
Add to Enqueued List
Yes
Examine first block in next row
Component or Connector?
Connector
Wire edges with assigned values?
No
Assign new value to all edges
Component
Yes
ground node?
No
Multiple Values?
No
Assign all edges the same value
Yes
Assign "0" to edge, replace existing assignment if necessary
Pick a value, assign all edges same value, replace all instances of conflicting value
Unassigned edges?
No
For each adjacent block not in enqueued list, add to stack and enqueued list
Yes
Pop stack
Generate new value, assign to an edge
Value?
No
Done
Yes