Multi-Core Beta Computer Appendix

Top_Module.v

```
// top-level Verilog module for 6.111 labkit (cjt, 4/07)

/* Quad-core Beta Computer Project, 6.111 Spring 2007
      Christopher Celio, c/o 2008
      Matt Long, c/o 2008

      plays the Game of Life, has two-preset playfields
      to load from ROM
*/

module beta2demo_labkit  (beep, audio_reset_b, ac97_sdata_out, ac97_sdata_in,
ac97_synch,
             ac97_bit_clock,

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

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

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

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

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

             clock_feedback_out, clock_feedback_in,

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

             rs232_txd, rs232_rxd, rs232_rts, rs232_cts,

             mouse_clock, mouse_data, keyboard_clock, keyboard_data,

             clock_27mhz, clock1, clock2,

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

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

             switch,

             led,

             user1, user2, user3, user4,
```

```
          daughtercard,

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

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

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

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

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

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

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

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

input  clock_feedback_in;
output clock_feedback_out;

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

output rs232_txd, rs232_rts;
input  rs232_rxd, rs232_cts;

input  mouse_clock, mouse_data, keyboard_clock, keyboard_data;

input  clock_27mhz, clock1, clock2;

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

input  button0, button1, button2, button3, button_enter, button_right,
```

```
     button_left, button_down, button_up;
input  [7:0] switch;
output [7:0] led;

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

inout [43:0] daughtercard;

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

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

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

// Audio Input and Output
assign beep= 1'b0;
assign audio_reset_b = 1'b0;
assign ac97_synch = 1'b0;
assign ac97_sdata_out = 1'b0;
// ac97_sdata_in is an input

// 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;
//assign disp_data_out = 1'b0;
// disp_data_in 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

 // 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;

 ////////////////////////////////////////////////////////////////////////
 //
 //  BETA2
 //
 ////////////////////////////////////////////////////////////////////////




 // VGA clock = 27mhz*7/6 = 31.5Mhz
 wire vclk_unbuf,vclk;
 DCM vclk1(.CLKIN(clock_27mhz),.CLKFX(vclk_unbuf));
 // synthesis attribute CLKFX_MULTIPLY of vclk1 is 7
 // synthesis attribute CLKFX_DIVIDE of vclk1 is 6
 // synthesis attribute CLK_FEEDBACK of vclk1 is NONE
 BUFG vclk2(.I(vclk_unbuf),.O(vclk));

 // processor clock = 27mhz*2/2 = 27MHz
 wire clk_unbuf,sys_clk;
 DCM clk1(.CLKIN(clock_27mhz),.CLKFX(clk_unbuf));
 // synthesis attribute CLKFX_MULTIPLY of clk1 is 2
 // synthesis attribute CLKFX_DIVIDE of clk1 is 2
 // synthesis attribute CLK_FEEDBACK of clk1 is NONE
 // synthesis attribute CLKIN_PERIOD of clk1 is 37
 BUFG clk2(.I(clk_unbuf),.O(sys_clk));

    reg raw_clk;
    wire clk;
    reg [31:0] count;
    wire [31:0] clock_period = 2 * switch[4];

    always @ (posedge sys_clk) begin

          if (count >= clock_period) begin
                raw_clk <= ~raw_clk; // flip clock signal
                count <= 32'b0; // reset counter
          end
          else begin
                count <= count + 1; // increment counter
          end

    end
```

```
      BUFGMUX BGM1 (.I0(sys_clk), .I1(raw_clk), .S(switch[5]), .O(clk));

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

   // tick_10ms is true for 1 clk cycle every 10ms
   reg tick_10ms;
   reg [18:0] tick_count;
   always @ (posedge clk) begin
      if (tick_count == 269999) begin
       tick_10ms <= 1;
       tick_count <= 0;
      end
      else begin
       tick_10ms <= 0;
       tick_count <= tick_count + 1;
      end
   end

   // reset circuitry: 16-cycle power-on reset and
   // a pushbutton for the user to use
   wire power_on_reset;
   SRL16 reset_sr (.D(1'b0), .CLK(clk), .Q(power_on_reset),
               .A0(1'b1), .A1(1'b1), .A2(1'b1), .A3(1'b1));
   defparam reset_sr.INIT = 16'hFFFF;
   wire user_reset, button_right_sync;
   debounce dbreset(clk,tick_10ms,~button_left,user_reset);
   debounce dbuttonr(clk,tick_10ms,~button_right, button_right_sync);
   wire reset = power_on_reset | user_reset;


//***********************************************
//***********************************************
//************  CELIO HARVARD BETA  **************
//***********************************************
//***********************************************

      /*wire enable_forward;
      button_push_enable step_forward(.clk(clk)
                                                                  , .reset(reset)
                                                                  ,
.button(button_right_sync)
                                                                  ,
.enable(enable_forward));
      */

      //used for debugging to Logic Analyzer
      wire [31:0] r0;

   //wire [31:0] beta_mdin0, beta_mdin1;        //, beta_mdout0, beta_mdout1;

      wire gol_busy;
      wire stall0, stall1, stall2, stall3;
      wire [5:0] opcode0, opcode1, opcode2, opcode3;
      wire [4:0] register_waddr0, register_waddr1, register_waddr2, register_waddr3;
      wire [31:0] beta_id0, beta_id1, beta_id2, beta_id3; //from inst_mem
      wire [31:0] beta_ia0, beta_ia1, beta_ia2, beta_ia3; //to inst_mem
   wire [31:0] beta_ma0, beta_ma1, beta_ma2, beta_ma3;

      wire [31:0] beta_mrd0, beta_mrd1, beta_mrd2, beta_mrd3; //into the beta
```

```
      wire [31:0] beta_mwd0, beta_mwd1, beta_mwd2, beta_mwd3; //out from the beta
      wire beta_moe0, beta_moe1, beta_moe2, beta_moe3;            //memory output
enables
      wire beta_mwe0, beta_mwe1, beta_mwe2, beta_mwe3;            //memory write
enables

      wire [31:0] ram_out0, ram_out1, ram_in0, ram_in1, ram_out2, ram_in2, ram_out3,
ram_in3;
      wire en_ram0, en_ram1, en_ram2, en_ram3;
      wire b_s_data0, b_s_data1, b_s_data2, b_s_data3;
      wire en_g_b_s0, en_g_b_s1, en_g_b_s2, en_g_b_s3;
      wire arbiter_moe0, arbiter_moe1, arbiter_moe2, arbiter_moe3;
      wire arbiter_wr0, arbiter_wr1, arbiter_wr2, arbiter_wr3;

   wire [30:0] irq_addr;
      wire irq;

      wire [7:0] core_count;
      assign core_count = {5'b0, switch[3:1]};

//*******************<<< CPU #0 >>>**************************

      BetaHarvardCelio CelioCPU0(.clk(clk)
                                                    ,.reset(reset | gol_busy)
                                                    ,.debug(debug)
                                                    ,.ext_stall(stall0)
                                                    ,.irq(irq)
                                                    ,.irq_addr(irq_addr)
                                                    ,.ia(beta_ia0)
                                                    ,.id(beta_id0)
                                                    ,.ma(beta_ma0)
                                                    ,.moe(beta_moe0)
                                                    ,.mrd(beta_mrd0)
                                                    ,.wr(beta_mwe0)
                                                    ,.mwd(beta_mwd0)
                                                    ,.cpu_id(8'd0)
      //hard-code each CPU ID#
                                                    ,.core_count(core_count)
                                                    //debugging
                                                    ,.r0(r0)
                                                    ,.opcode(opcode0)
                                                    ,.waddr(register_waddr0)
                                                    );

//*******************<<< CPU #1 >>>**************************

      BetaHarvardCelio CelioCPU1(.clk(clk)
                                                    ,.reset(reset | gol_busy)
                                                    ,.debug(debug)
                                                    ,.ext_stall(stall1)
                                                    ,.irq(irq)
                                                    ,.irq_addr(irq_addr)
                                                    ,.ia(beta_ia1)
                                                    ,.id(beta_id1)
                                                    ,.ma(beta_ma1)
                                                    ,.moe(beta_moe1)
                                                    ,.mrd(beta_mrd1)
                                                    ,.wr(beta_mwe1)
                                                    ,.mwd(beta_mwd1)
```

```
                                        ,.cpu_id(8'd1)
      //hard-code each CPU ID#
                                        ,.core_count(core_count)

                                        //,.r0(r0)
                                        ,.opcode(opcode1)
                                        ,.waddr(register_waddr1)
                                        );

//****************<<< CPU #2 >>>*************************

      BetaHarvardCelio CelioCPU2(.clk(clk)
                                        ,.reset(reset | gol_busy)
                                        ,.debug(debug)
                                        ,.ext_stall(stall2)
                                        ,.irq(irq)
                                        ,.irq_addr(irq_addr)
                                        ,.ia(beta_ia2)
                                        ,.id(beta_id2)
                                        ,.ma(beta_ma2)
                                        ,.moe(beta_moe2)
                                        ,.mrd(beta_mrd2)
                                        ,.wr(beta_mwe2)
                                        ,.mwd(beta_mwd2)
                                        ,.cpu_id(8'd2)
      //hard-code each CPU ID#
                                        ,.core_count(core_count)

                                        //,.r0(r0)
                                        ,.opcode(opcode2)
                                        ,.waddr(register_waddr2)
                                        );

//****************<<< CPU #3 >>>*************************

      BetaHarvardCelio CelioCPU3(.clk(clk)
                                        ,.reset(reset | gol_busy)
                                        ,.debug(debug)
                                        ,.ext_stall(stall3)
                                        ,.irq(irq)
                                        ,.irq_addr(irq_addr)
                                        ,.ia(beta_ia3)
                                        ,.id(beta_id3)
                                        ,.ma(beta_ma3)
                                        ,.moe(beta_moe3)
                                        ,.mrd(beta_mrd3)
                                        ,.wr(beta_mwe3)
                                        ,.mwd(beta_mwd3)
                                        ,.cpu_id(8'd3)
      //hard-code each CPU ID#
                                        ,.core_count(core_count)

                                        //,.r0(r0)
                                        ,.opcode(opcode3)
                                        ,.waddr(register_waddr3)
                                        );

//**********************************************************
//*************** MEMORY & MEMORY DECODING ****************
```

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

	wire en_ram, rd_ps2, en_char, en_gol; //, en_g_b_s;
	wire sel_ram, sel_char, sel_timer, sel_ps2, sel_gol_a, sel_gol_b;
	wire [31:0] char_out, ps2_out, gol_out;
	wire golram_select; //ALL Betas need direct access to this

	wire [31:0] mdin;

	ram_decoder ram_decoder0(.clk(clk)

	,.beta_ma(beta_ma0),.beta_mwd(beta_mwd0)

	,.beta_mrd(beta_mrd0),.beta_moe(beta_moe0),.beta_we(beta_mwe0)
					,.en_ram(en_ram0) //,.ram_ma(
					,.ram_in(ram_in0)
					,.ram_out(ram_out0)
					//arbiter
					,.moe(arbiter_moe0)
					,.we(arbiter_wr0) //,.ma( //,.mwd(
					,.mdin(mdin)
					,.gol_sel_out(golram_select)

	,.b_s_data(b_s_data0),.en_g_b_s(en_g_b_s0)
					);

	ram_decoder ram_decoder1(.clk(clk)

	,.beta_ma(beta_ma1),.beta_mwd(beta_mwd1)

	,.beta_mrd(beta_mrd1),.beta_moe(beta_moe1),.beta_we(beta_mwe1)
					,.en_ram(en_ram1) //,.ram_ma(
					,.ram_in(ram_in1)
					,.ram_out(ram_out1)
					//arbiter
					,.moe(arbiter_moe1)
					,.we(arbiter_wr1) //,.ma( //,.mwd(
					,.mdin(mdin)
					,.gol_sel_out(golram_select)

	,.b_s_data(b_s_data1),.en_g_b_s(en_g_b_s1)
					);

	ram_decoder ram_decoder2(.clk(clk)

	,.beta_ma(beta_ma2),.beta_mwd(beta_mwd2)

	,.beta_mrd(beta_mrd2),.beta_moe(beta_moe2),.beta_we(beta_mwe2)
					,.en_ram(en_ram2) //,.ram_ma(
					,.ram_in(ram_in2)
					,.ram_out(ram_out2)
					//arbiter
					,.moe(arbiter_moe2)
					,.we(arbiter_wr2) //,.ma( //,.mwd(
					,.mdin(mdin)
					,.gol_sel_out(golram_select)

	,.b_s_data(b_s_data2),.en_g_b_s(en_g_b_s2)
					);
```

```
ram_decoder ram_decoder3(.clk(clk)

,.beta_ma(beta_ma3),.beta_mwd(beta_mwd3)

,.beta_mrd(beta_mrd3),.beta_moe(beta_moe3),.beta_we(beta_mwe3)
                                          ,.en_ram(en_ram3) //,.ram_ma(
                                          ,.ram_in(ram_in3)
                                          ,.ram_out(ram_out3)
                                          //arbiter
                                          ,.moe(arbiter_moe3)
                                          ,.we(arbiter_wr3) //,.ma( //,.mwd(
                                          ,.mdin(mdin)
                                          ,.gol_sel_out(golram_select)

,.b_s_data(b_s_data3),.en_g_b_s(en_g_b_s3)
                                          );

// private memory for each core: up to 16K x 32
Beta_OS_203f0test ram0(beta_ma0[15:2], clk, ram_in0, ram_out0, en_ram0);
Beta_OS_203f8test ram1(beta_ma1[15:2], clk, ram_in1, ram_out1, en_ram1);
Beta_OS_203f16test ram2(beta_ma2[15:2], clk, ram_in2, ram_out2, en_ram2);
Beta_OS_203f24test ram3(beta_ma3[15:2], clk, ram_in3, ram_out3, en_ram3);

// private instruction memory: up to 16K x 32
// (memory module generated by betamem.py)
//addrA, addrB, clk, dinA, dinB, doutA, doutB, weA, weB)
//Beta_OS_202k0test inst_mem0(beta_ia0[15:2], beta_ia1[15:2], clk, 32'hFFFFFFFF,
//32'hFFFFFFFF, beta_id0,     beta_id1, 1'b0, 1'b0);
Beta_OS_203f32test inst_mem0(beta_ia0[15:2], clk, 32'hFFFFFFFF, beta_id0, 1'b0);
Beta_OS_203f40test inst_mem1(beta_ia1[15:2], clk, 32'hFFFFFFFF, beta_id1, 1'b0);
Beta_OS_203f48test inst_mem2(beta_ia2[15:2], clk, 32'hFFFFFFFF, beta_id2, 1'b0);
Beta_OS_203f56test inst_mem3(beta_ia3[15:2], clk, 32'hFFFFFFFF, beta_id3, 1'b0);

wire [31:0] ma_arbiter_out, mwd_arbiter_out;
wire wr_arbiter_out, moe_arbiter_out;

memory_arbiter_quadcore arbiter_v2(.clk(clk)

,.debug(1'b0/*switch[1]*/)

,.ma3(beta_ma3),.ma2(beta_ma2),.ma1(beta_ma1),.ma0(beta_ma0)

,.moe({arbiter_moe3,arbiter_moe2,arbiter_moe1,arbiter_moe0})

,.mwd3(beta_mwd3),.mwd2(beta_mwd2),.mwd1(beta_mwd1),.mwd0(beta_mwd0)

,.wr({arbiter_wr3,arbiter_wr2,arbiter_wr1,arbiter_wr0})

,.stall({stall3,stall2,stall1, stall0})

,.ma_out(ma_arbiter_out)

,.mwd_out(mwd_arbiter_out)

,.wr_out(wr_arbiter_out)

,.moe_out(moe_arbiter_out));
```

```
      mem_decoder shared_mem_decoder
                                          (     .clk(clk),
                                          .mwe(wr_arbiter_out), .moe(moe_arbiter_out),
.maddr(ma_arbiter_out)
                                          , .mdin(mdin)
                                          ,.en_ram(en_ram)
                                          , .rd_ps2(rd_ps2)
                                          , .en_char(en_char), .en_gol(en_gol)
                                          , .en_sync(en_sync),
                                          .sel_ram(sel_ram), .sel_char(sel_char),
.sel_timer(sel_timer),
                                          .sel_ps2(sel_ps2), .sel_gol_a(sel_gol_a),
.sel_gol_b(sel_gol_b),
                                          .sync_out(sync_out),
                                          .ram_out(ram_out), .ps2_out(ps2_out),
.char_out(char_out), .gol_out(gol_out)
                                          ,.golsel_out(golram_select)
                                          );


      //small amount of shared RAM for synchronizing cores
      //holds an array, each core "checks in" when it finishes a task, then waits
      //core#0 waits for all cores to check in, then notifies all cores to proceed to
next step

      wire [63:0] sync_reg_out;
      synch_ram sync_ram(.clk(clk)
                                                ,.reset(reset)
                                                ,.debug(debug)
                                                ,.mem_addr(ma_arbiter_out)
                                                ,.data_in(mwd_arbiter_out)
                                                ,.data_out(sync_out)
                                                ,.enable(en_sync)
                                                ,.sync_out(sync_reg_out)
                                                );

//************************************************************
//****************** DISPLAY STUFF ***********************
//************************************************************

      assign vga_out_pixel_clock = ~vclk;  // give data time to arrive
      assign vga_out_sync_b = 1'b1; // not used

      wire [13:0] cellnum;
      wire cell_state;
      wire [11:0] char_addr;
      wire [7:0] char_code;


      //added today
      //wire [10:0] font_addr;
      //wire [7:0] font_byte;

      //calculate memory_addresses for GameOfLife buffer access
      wire [31:0] gola_offset = ma_arbiter_out - 32'hFFFF_8d00;
      wire [31:0] golb_offset = ma_arbiter_out - 32'hFFFF_c600;
      wire [13:0] addrb = sel_gol_b ? golb_offset[13:0] : gola_offset[13:0];

      assign mit_logo = switch[6];
```

```
      vga_m vgam( .pix_clk(vclk),
                              .reset(reset),
                              .hsout(vga_out_hsync), .vsout(vga_out_vsync),
                              .blank_b(vga_out_blank_b),
                              .color({vga_out_red, vga_out_green, vga_out_blue}),
                              .mode(display_mode),
                              .cellnum(cellnum), .cell_state(cell_state),
.char_addr(char_addr), .char_code(char_code),
                              .addrb(addrb), .sel_gol(sel_gol_a | sel_gol_b),
.mit_logo(mit_logo)

                  );

      //pulled out of td1
      //font f(.addr(font_addr), .clk(vclk), .row(font_byte));  //The FONT ROM

      cmem cm1(   .addra(char_addr), .clka(~vclk), .douta(char_code),
                              .addrb(ma_arbiter_out[11:2]), .clkb(clk)
                              ,.dinb(mwd_arbiter_out), .doutb(char_out), .web(en_char)
                  );

      //logic for GoL buffer select register
      wire en_g_b_s, g_b_s_wdata;
      assign en_g_b_s = en_g_b_s0; //| en_g_b_s1;
      assign g_b_s_wdata = b_s_data0; //| b_s_data1;

      wire rom_select = switch[7];

      gol_mem_manager gmm1(.addra(cellnum), .clka(vclk), .douta(cell_state),
            .addrb(addrb), .clkb(clk), .dinb(mwd_arbiter_out), .doutb(gol_out),
.web(en_gol),
            .sel_gol_a(sel_gol_a), .sel_gol_b(sel_gol_b)
            , .en_g_b_s(en_g_b_s),
.golram_select(golram_select),.g_b_s_wdata(g_b_s_wdata)
            , .debug(1'b0/*switch[3]*/)
            , .reset(reset), .busy(gol_busy), .rom_sel(rom_select)
                  );

//**********************************************************
//**********  INTERFACING & OTHER ODDS&ENDS  ***************
//**********************************************************

   // ps2 interface
   ps2 kbd(clk,reset,tick_10ms,keyboard_clock,keyboard_data,
           rd_ps2,ps2_out[7:0],ps2_out[8],ps2_out[9], {1'b0, core_count});
   assign ps2_out[31:10] = 0;

   // timer interrupt
   reg irq_timer;
   always @ (posedge clk) begin
      if (sel_timer || reset) irq_timer <= 0;
      else if (tick_10ms) irq_timer <= 1;
   end

   // interrupts
   assign irq = ~ps2_out[8] || irq_timer;
   assign irq_addr = ~ps2_out[8] ? 12 : 8;
```

```
	//count #rounds
	reg [15:0] rounds;
	always @ (posedge golram_select) begin
		if(reset) rounds <= 0;
		else rounds <= (rounds + 1);
	end

	// count # rounds/per second
	reg [15:0] rounds_per_second;
	reg [15:0] last_round_count;
	reg [31:0] cycle_count;
	always @ (posedge sys_clk) begin
	if (cycle_count >= 26999999) begin
			rounds_per_second <= rounds - last_round_count;
			last_round_count <= rounds;
			cycle_count <= 0;
		end
		else begin
			cycle_count <= cycle_count + 1;
		end
	end

   // display mem addr
	wire [63:0] display_numbers;
	assign display_numbers = {rounds, rounds_per_second, beta_ia1[15:0],
beta_ia0[15:0]};
	//assign display_numbers = switch[6] ? sync_reg_out : {rounds, rounds_per_second,
beta_ia1[15:0], beta_ia0[15:0]};
	//assign display_numbers[63:32] = switch[5] ? beta_ia0 : beta_id1;
	//assign display_numbers[31:0] = switch[4] ? beta_ia1 : beta_id0;

   // display mem addr
   display_16hex disp(reset, clock_27mhz, display_numbers,
		disp_blank, disp_clock, disp_rs, disp_ce_b,
		disp_reset_b, disp_data_out);

	//LEDs (inverse logic)
	assign led = ~{golram_select, reset, clk, irq, 4'b0};

	//Switch Zero used to control display mode (Terminal or Game of Life mode)
	assign display_mode = switch[0];

	/*********************************************************/
	//debugging extravaganza!

	assign analyzer1_clock = clk;

	assign analyzer1_data = {clk, beta_mwe0, beta_mwe1, beta_mwe2, beta_moe0,
beta_moe1,
							beta_moe2, wr_arbiter_out,
moe_arbiter_out,
							golram_select, sel_gol_a,
sel_gol_b, 3'b0, reset};

	assign analyzer2_data = {opcode0[5:0], opcode1[5:0], stall0, stall1, stall2,
1'b0};

	assign analyzer3_data = {ma_arbiter_out[7:0], mdin[5:0], core_count[1:0]};
```

```
      assign analyzer4_data = {beta_ma0[7:0], beta_ma1[7:0]};
endmodule
```

Beta.v

```
////////////////////////////////////////////////////////////////////////////////
// Company:        6.111 Spring 2007
// Engineer:       Christopher Celio, EECS, c/o 2008
//
// Create Date:    16:45:29 04/11/2007
// Design Name:    One-stage Pipelined Beta Processor
// Module Name:    Beta
// Project Name:   Multi-core Beta Computer
// Target Devices: Xilinx XC2v6000 Virtex FPGA
// Tool versions:
// Description:    This is the Beta processor interface,
//                                  that connects the PC unit, RegFile, ALU,
//                                  Control_Unit.
//                 Two_stage pipeline crashes: it somehow jumps to a location in stack
memory
//                 and then types out an illegal operation to the character buffer.
which is wierd,
//                 b/c the pipelined beta works well enough to write to the screen
correctly, jmp
//                 to execute illegal_ops correctly, and a host of other tasks.  The
error might be in a
//                 failure to LD/LDR correctly (the source of most of my errors).
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
//                       0.02 - added "get_cpu_id#" and "get_core_count" lines to WDSEL
mux
//                                          very useful for multi-core software
//
// Additional Comments:
//                 (not true of the working version below)
//          Two-stage Pipelined, broken into Inst_Fetch and Execution Phase
//
//
//          the code is directly inspired by a modified beta, taken from the file
//          'labproject_pipelinedbeta'    created by Chris Celio in
//          May 2006 as a final project for 6.004: Computation Structures,
//          which was based on 'lab6beta' created earlier in the class.
//
//          The Beta processor is used as a teaching tool in 6.004, which
//          students build as part of the course.  Therefore, much help in
//          constructing this 6.111 Beta has been received through old course
//          notes, video lectures, and discussion with Prof. Chris Terman.
//
//
//
//
////////////////////////////////////////////////////////////////////////////////
module BetaHarvardCelio(clk, reset, debug, ext_stall, irq, irq_addr, ia, id, ma, moe,
mrd, wr, mwd
          , sup_bit, r0, opcode, waddr, cpu_id, core_count);
    input clk;                      //clock signal, entire system in synchronized on
this
    input reset;              //reset system, start reading at IA=32'h80000000;
    input [7:0] debug;  //4bit debug signal, no use right now.....
       input ext_stall;             //stalls pipeline if waiting for memory data, etc.,
passes to CTL
    input irq;                      //Interupt Request
```

```
      input [30:0] irq_addr; //Interrupt Address. ie, I_Clk, I_Kbd, etc. MUST set
supervisor bit
    output [31:0] ia;   //Instruction Address
    input [31:0] id;          //Instruction Data
    output [31:0] ma;   //Memory Address
    output moe;               //Memory Output Enable
    input [31:0] mrd;   //Memory Read Data
    output wr;                      //Memory Write Enable
    output [31:0] mwd;  //Memory Write Data
       input [7:0] cpu_id; //hard-wired CPU_ID
       input [7:0] core_count; //core_count, this may be changed by debug_switches

       //debugging outs to the Logic Analyzer
       output sup_bit; //supervisor bit....
       output [31:0] r0;
       output [5:0] opcode;
       output [4:0] waddr;


//*****************************************
//*****************************************
//**********    MAIN BLOCKS    ************
//*****************************************
//*****************************************

//Parameters
parameter XP_REG = 5'd30;
parameter NOP = {6'b100_000, 15'b11111_11111_11111, 11'b0}; //ADD(R31,R31,R31)

//registers (registered pc_if is inside the pc module, and reg_file is the last bit of
registers).
//reg [31:0] pc_exe = 0;
//reg [31:0] id_exe = 0;      //i *hate* verilog. this was the cause of quite a bug in
simulation...

wire [31:0] pc_exe, id_exe;

//control signals
wire Z, werf, asel, bsel, ra2sel, wasel, stall;
wire [5:0] alufn;
wire [2:0] wdsel;
wire [2:0] pcsel;

//busses
wire annul_if;                     //annulling Inst_Data
wire [31:0] id_muxed;

wire [31:0] bro;        //branch address
wire [31:0] jt;         //jump address
wire [31:0] aseldata; //ASEL mux
wire [31:0] radata;
wire [31:0] bseldata;   //BSEL mux
wire [31:0] wdata;      //write data, WDSEL mux
wire [31:0] pc_sum; //output from PC, goes to PC_EXE register




//*****************************************
//**********     IF STAGE      ************
//*****************************************

pc          X_pc_if(.clk(clk)
                  ,.reset(reset)
                  ,.stall(stall)
                  ,.pcsel(pcsel[2:0])
                  ,.inst_addr(ia[31:0])
```

```
                  ,.sum_addr(pc_sum[31:0])
                  ,.jt_addr(jt[31:0])
                  ,.bro_addr(bro[31:0])
                  ,.xaddr(irq_addr[30:0]));


//*********************
//** Branching Annulment
/*
assign annul_if = reset | (|pcsel); //if pcsel != 0, then annul id_exe
assign id_muxed = annul_if ? NOP : id;


//****************************************
//********   Pipelined Registers   ********
//****************************************

always @ (posedge clk) begin

      pc_exe <= stall ? pc_exe : {pc_sum[31:2], 2'b0};
      id_exe <= stall ? id_exe : id_muxed; //reset ? NOP :
                  //id_muxed; //make sure no instructions are executed while RESET is
high

end
*/


assign pc_exe = stall ? pc_exe : {pc_sum[31:2], 2'b0};
assign id_exe = stall ? id_exe : id;

//    assign pc_exe = {pc_sum[31:2], 2'b0};
//    assign id_exe = id;
//
//****************************************
//*********     EXE STAGE     ************
//****************************************

//module register_file(clk, reset, werf, raddr1, rdata1, raddr2, rdata2,waddr, wdata);

//wire ra2sel;
wire [4:0] raddr2;
assign raddr2 = ra2sel ? id_exe[25:21] : id_exe[15:11];

//wire wasel;
//wire [4:0] waddr;
assign waddr = wasel ? XP_REG : id_exe[25:21];

register_file     X_regfile(.clk(clk)
                  ,.reset(reset)
                  ,.werf(werf)                  //Write_Enable Reg. File
                  ,.raddr1(id_exe[20:16]) //ra[4:0]
                  ,.rdata1(radata[31:0])
                  ,.raddr2(raddr2) //rb[4:0]
                  ,.rdata2(mwd[31:0])
                  ,.waddr(waddr) //rc[4:0]
                  ,.wdata(wdata[31:0])
                  ,.r0(r0));

control_logic     X_ctl(  .clk(clk)
                  , .reset(reset)  //makes sure wr is low so we don't rewrite anything
                  , .opcode(id_exe[31:26]) //OPCODE
                  , .sup_bit(pc_exe[31])        //supervisor bit BUG, needs to be
pc_ia_exe[31]
                  , .IRQ(irq)                   //Interupt Request
                  , .ext_stall(ext_stall)
                  , .Z(Z)                             //Branching logic
```

```
                  , .alufn(alufn[5:0])    //ALU function
                  , .moe(moe)                          //Memory Output Enable
                  , .werf(werf)                  //Write_Enable Register File
                  , .bsel(bsel)                  //select b input to ALU
                  , .wdsel(wdsel)   //Write_Data selector (ALU, PC+4, Memory_Data?)
                  , .wr(wr)                      //Data Memory Write/Read(?)
                  , .ra2sel(ra2sel)       //select Addr for Rb read port (Reg_File)
                  , .pcsel(pcsel[2:0])    //Program Counter selector (Xadr,Ill_op,JT,
bro, sum)
                  , .asel(asel)                  //select a input to ALU
                  , .wasel(wasel)
                  , .stall(stall));       //Write_Addr selector for Reg_File


      assign sup_bit = pc_exe[31]; //is this really the supervisor bit?
      assign opcode = id_exe[31:26];

//selects XP as destination Reg. during exceptions

alu         X_alu(.alufn(alufn[5:0])
                  ,.a(aseldata[31:0])
                  ,.b(bseldata[31:0])
                  ,.out(ma[31:0]) //output
                  );



      //*****************
      //*** SEXT, BRANCH-OFFEST  ***
      // (PC+4) + 4*SXT(C)
      assign bro = pc_exe + {{14{id_exe[15]}}, id_exe[15:0], 2'b0}; //addresses done as
byte addresses

      //*****************
      //*** ASEL MUX  ***

      //this is the A input to the ALU , used for LDR
      assign aseldata = asel ? {1'b0, bro[30:0]} : radata;

      //*****************
      //*** BSEL MUX  ***

      //this is the B input to the ALU, choose between either Reg_File Rb, or Literal
from Inst_data
      //BUG is the ordering of this correct and asel correct?
      assign bseldata = bsel ? {{16{id_exe[15]}}, id_exe[15:0]} : mwd;  //mwd is the
RB_Data from Reg_File

      //*****************
      //***  Z LOGIC  ***
      assign Z = ~|radata; //NOR all 31 inputs, ie, is radata ALL zeroes?

      //*****************
      //***  JT_ADDR  ***

      assign jt = radata;

      //*****************
      //*** WDSEL MUX ***
      assign wdata = (wdsel==2) ? mrd :
                  (wdsel==1) ? ma  :
                  (wdsel==3) ?  {24'b0, cpu_id} : //CPU_ID, CORE_COUNT are 8-bit
numbers to allow up to 255 cores
                  (wdsel==4) ?  {24'b0, core_count} :
                  {pc_exe[31], pc_exe[30:0]};
```

```
//****************************************
//****************************************
endmodule
```

PC.v

```
//////////////////////////////////////////////////////////////////////////////////
// Company:       6.111 Spring 2007
// Engineer:      Christopher Celio, EECS, c/o 2008
//
// Create Date:    17:11:09 04/11/2007
// Design Name:
// Module Name:    pc
// Project Name:
// Target Devices:
// Tool versions:
// Description: Program Counter for the Beta Processor
//                                  its job is to deduce the next Instruction Address
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module pc(clk, reset, stall, pcsel, inst_addr, sum_addr, jt_addr, bro_addr, xaddr);
    input clk;                      //clock
    input reset;              //reset
       input stall; //freeze the PC counter to stall the pipeline
    input [2:0] pcsel;  //selector choices among five choices for the inst_addr
    output [31:0] inst_addr;  //instruction address, the output, this is registered by
the BRAMs, and not here!
    output [31:0] sum_addr;   //simply the last inst_addr + 4, sent to PC_EXE register
    input [31:0] jt_addr;           //Jump_to instr_addr, given by register_addr_1
    input [31:0] bro_addr;    //branch_addr, computed during BEQ,BNE ops
       input [30:0] xaddr; //exception addr, e.g., Irq_Clk, Irq_Kbd, Irq_Mouse, etc.

parameter ILL_OP_ADDR = 32'h80000004; //TRAP address
parameter RESET_ADDR =  32'h80000000; //RESET address

reg [31:0] inst_addr_reg;
//this is the internal memory of the PC (==PC_EXE, but +4 of the IA given to the BRAMs)

assign sum_addr = inst_addr_reg + 4;

assign inst_addr = reset ? RESET_ADDR : //ia_muxed = reset ? RESET_ADDR :
                                    stall ? inst_addr_reg :
                                    (pcsel==3'd4) ? {1'b1, xaddr} :
//{1'b1,xaddr[30:0]} :  //interupt_addr
                                    (pcsel==3'd3) ? ILL_OP_ADDR : //trap
                                    (pcsel==3'd2) ? {inst_addr_reg[31] & jt_addr[31],
jt_addr[30:2], 2'b00} : //jump to , can clear sup-bit
                                    (pcsel==3'd1) ? {inst_addr_reg[31],bro_addr[30:2],
2'b00} : //branch
                                    //(pcsel==3'd0)
                                          {inst_addr_reg[31], sum_addr[30:0]};

always @ (posedge clk) begin
      inst_addr_reg <= {inst_addr[31:2], 2'b0};
end

endmodule
```

Register_File.v

```
//////////////////////////////////////////////////////////////////////////////////
// Company:       6.111 Spring 2007
```

```
// Engineer:      Christopher Celio, EECS, c/o 2008
//
// Create Date:    18:33:12 04/16/2007
// Design Name:
// Module Name:    register_file
// Project Name:
// Target Devices:
// Tool versions:
// Description: Register File for Beta Processor
//                          stores 31 values in distributed memory.
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments: Code ported from Celio's 6.004 Beta, with
//                      bug squashing help from Prof. Terman
//
////////////////////////////////////////////////////////////////////////////////
//2-read ports, 1-write port.
//32 registers, each 32-bit words. well, actually only 31 registers, there is no R31
//R31 always equals 0
module register_file(clk, reset, werf, raddr1, rdata1, raddr2, rdata2,waddr, wdata,
r0);

      input clk;                    //set this to the beta's system_clock
      input reset;                  //unused, but could be used to clear memory
      input werf;                   //Write Enable Register File
      input [4:0] raddr1; //address for read port (Reg[A])
      input [4:0] raddr2; //address for read port (Reg[B])
      input [4:0] waddr; //write address port (Reg[C])

      //for debugging.....
      output [31:0] r0;

      output [31:0] rdata1,rdata2; //read data
      input [31:0] wdata;                 //write data

  (* ram_style = "distributed" *) //trick learned from Terman
                                                                  //since new verilog
is buggy at implicit ram definitions
      reg [31:0] registers[31:0]; //31 registers, as there is no R31


      assign rdata1 = registers[raddr1];
      assign rdata2 = registers[raddr2];

      assign r0 = registers[0];

      //write port, if WERF (Write Enable Register File) high
      always @(posedge clk) begin
                  registers[waddr] <= (werf && waddr!= 31) ? wdata : registers[waddr];
      end

endmodule
```

Control_Logic.v

```
////////////////////////////////////////////////////////////////////////////////
// Company:       6.111 Spring 2007
// Engineer:      Christopher Celio, EECS, c/o 2008
//                          patner Matt Long, EECS, c/o 2008
//
// Create Date:    18:21:08 04/16/2007
// Design Name:
// Module Name:    control_logic
```

```verilog
// Project Name:
// Target Devices:
// Tool versions:
// Description: Takes in Beta opcode, IRQ, and Z logic
//                                    to deduce all control signals for Beta CPU
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:  created using 6.004 lecture notes
//
////////////////////////////////////////////////////////////////////////////////
module control_logic(clk
                                                  , reset
                                                  , opcode
                                                  , sup_bit
                                                  , ext_stall
                                                  , IRQ
                                                  , Z
                                                  , alufn
                                                  , moe
                                                  , werf
                                                  , bsel
                                                  , wdsel
                                                  , wr
                                                  , ra2sel
                                                  , pcsel
                                                  , asel
                                                  , wasel
                                                  , stall);

      //input operation code
      input [5:0] opcode;

      input clk;
      input sup_bit; //supervisor bit
      input reset;
      input ext_stall; //external module (from Beta) telling the Beta to stall
      input IRQ;
      input Z; //used for BEQ, BNE branching logic

      //output control signals
      output [5:0] alufn;
      output moe;             //memory output enable. hi only for LD, LDR
      output werf;
      output bsel;
      output [2:0] wdsel;
      output wr;
      output ra2sel;
      output [2:0] pcsel;
      output asel;
      output wasel;
      output stall; //either stalls the processor if ext_stall or doing a LD

      wire st, ld, ldr, interrupt;
      reg illop_trap;
      reg moe_temp;
      //reg stall_temp;
      reg load_stalled; //the pipeline is *currently* stalled due to a LOAD operation
      reg [2:0] wdsel_temp;
      reg [5:0] alufn;


      assign st = (opcode==6'b011_001);
      assign ld = (opcode==6'b011_000);
      assign ldr = (opcode==6'b011_111);
```

```
      assign interrupt = sup_bit ? 0: IRQ; //do not assert IRQ if already in supervisor
mode

      assign werf       = (st & !interrupt & !illop_trap) ? 0 : 1; //(only zero for
"ST")
      assign bsel       = opcode[4]; //(only zero for "OP", one for "OPC","LD","ST")

      assign wdsel = (interrupt | illop_trap) ? 0 : wdsel_temp; //to make sure that
interrupts override


      assign wr         = (st & !reset & !interrupt & !illop_trap) ? 1 : 0; //((only
one for "ST")
      assign ra2sel     = st ? 1 : 0; //(only one for "ST")


      //TODO, move this to the always loop, probably makes this faster?
      assign pcsel = illop_trap ? 3'h3 :
                              interrupt ? 3'h4 :
                              (opcode==6'b011_101) ? ( Z ? 3'h1 : 3'h0) : //beq
                              (opcode==6'b011_110) ? ( Z ? 3'h0 : 3'h1) : //bne
                              (opcode==6'b011_011) ? 3'h2 : //jump
                              0; //default

      assign asel = ldr ? 1 : 0; //(only one for "LDR")
      assign wasel = (interrupt | illop_trap) ? 1 : 0; //(don't care for ST)

      assign moe = (interrupt | illop_trap) ? 0 : moe_temp;

      wire load_stall; //, prev_load_stalled;
      reg prev_load_stalled, prev_ext_stalled,prev_load_stalled_temp,
prev_ext_stalled_temp;
      //assign stall = (ext_stall |ldr | ld | st) && ~load_stalled;
      //assign stall = ext_stall | ((ldr | ld) && ~load_stalled);


      //always @ (posedge clk) load_stalled <= (ldr | ld) & ~load_stalled;

      //*********************************************************
      always @ (posedge clk) prev_load_stalled <= load_stall;
      always @ (posedge clk) prev_ext_stalled <= ext_stall;

      assign load_stall = ~ext_stall & (ldr | ld | st) & ~prev_load_stalled;

      assign stall = ext_stall
                        | (prev_ext_stalled & (ldr | ld | st))
                        | (load_stall);

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

      always @ * begin

            //initialize all signals
            alufn = 6'hx;
            wdsel_temp = 3'h0;
            moe_temp = 1'b0;
            illop_trap = 1'b0;

            case(opcode)

                  6'b011_000: begin //LD
                        alufn = 6'b000_000; //"+"
                        wdsel_temp = 3'h2;
                        moe_temp = 1'b1;
                  end
                  6'b011_001: begin //ST
```

```
      alufn = 6'b000_000; //"+"
      wdsel_temp = 3'hx;
end
6'b011_011: begin //JMP
end
6'b011_101: begin //BEQ
end
6'b011_110: begin //BNE
end
6'b011_111: begin //LDR
      alufn = 6'h1a; //"A"
      wdsel_temp = 3'h2;
      moe_temp = 1'b1;
end
6'b100_000: begin //ADD
      alufn = 6'h00; //"+"
      wdsel_temp = 3'h1;
end
6'b100_001: begin //SUB
      alufn = 6'h01; //"-"
      wdsel_temp = 3'h1;
end
6'b100_010: begin //MUL
      alufn = 6'b000_010; //"*"
      wdsel_temp = 3'h1;
end
6'b100_011: begin //DIV
      alufn = 6'b000_100; //"/"
      wdsel_temp = 3'h1;
end
6'b100_100: begin //CMPEQ
      alufn = 6'h33;
      wdsel_temp = 3'h1;
end
6'b100_101: begin //CMPLT
      alufn = 6'h35;
      wdsel_temp = 3'h1;
end
6'b100_110: begin //CMPLE
      alufn = 6'h37;
      wdsel_temp = 3'h1;
end
6'b101_000:begin //AND
      alufn = 6'h18;
      wdsel_temp = 3'h1;
end
6'b101_001:begin //OR
      alufn = 6'h1e;
      wdsel_temp = 3'h1;
end
6'b101_010:begin //XOR
      alufn = 6'h16;
      wdsel_temp = 3'h1;
end
6'b101_100:begin //SHL
      alufn = 6'h20;
      wdsel_temp = 3'h1;
end
6'b101_101:begin //SHR
      alufn = 6'h21;
      wdsel_temp = 3'h1;
end
6'b101_110:begin //SRA
      alufn = 6'h23;
      wdsel_temp = 3'h1;
end
```

```
        6'b110_000:begin //ADDC
              alufn = 6'h00;
              wdsel_temp = 3'h1;
        end
        6'b110_001:begin //SUBC
              alufn = 6'h01;
              wdsel_temp = 3'h1;
        end
        6'b110_010:begin //MULC
              alufn = 6'b000_010;
              wdsel_temp = 3'h1;
        end
        6'b110_011:begin //DIVC
              alufn = 6'b000_100;
              wdsel_temp = 3'h1;
        end
        6'b110_100:begin //CMPEQC
              alufn = 6'h33;
              wdsel_temp = 3'h1;
        end
        6'b110_101:begin //CMPLTC
              alufn = 6'h35;
              wdsel_temp = 3'h1;
        end
        6'b110_110:begin //CMPLEC
              alufn = 6'h37;
              wdsel_temp = 3'h1;
        end

        6'b111_000:begin //ANDC
              alufn = 6'h18;
              wdsel_temp = 3'h1;
        end
        6'b111_001:begin //ORC
              alufn = 6'h1e;
              wdsel_temp = 3'h1;
        end
        6'b111_010:begin //XORC
              alufn = 6'h16;
              wdsel_temp = 3'h1;
        end
        6'b111_100:begin //SHLC
              alufn = 6'h20;
              wdsel_temp = 3'h1;
        end
        6'b111_101:begin //SHRC
              alufn = 6'h21;
              wdsel_temp = 3'h1;
        end
        6'b111_110:begin //SRAC
              alufn = 6'h23;
              wdsel_temp = 3'h1;
        end
        6'b101_011: begin //GET CPU_ID
              wdsel_temp = 3'h3;
        end
        6'b101_111: begin //GET CORE_COUNT
              wdsel_temp = 3'h4;
        end



        default: begin //ILLEGAL OPERATION
              illop_trap = 1'b1;
        end

  endcase
```

```
      end

endmodule
```

ALU.v

```
/////////////////////////////////////////////////////////////////////////////////
// Company:       6.111 Spring 2007
// Engineer:      Christopher Celio, EECS, c/o 2008
//
// Create Date:    21:05:36 04/11/2007
// Design Name:
// Module Name:    alu
// Project Name:
// Target Devices:
// Tool versions:
// Description: Arithmetic Logic Unit for Beta processor
//
//
// Dependencies: shift_right.v
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments: built with help from 6.004 notes and Chris Celio's 6.004_lab3
ALU
//
//
/////////////////////////////////////////////////////////////////////////////////
module alu(alufn, a, b, out); //, z, v, n
    input [5:0] alufn;        //function for ALU to perform
    input signed [31:0] a;    //input 1
    input signed [31:0] b;    //input 2
    output signed [31:0] out; //output of ALU
 //   output alu_z;           //a == b //true when ALL bits are zero
 //   output alu_v;           //detects overflows?
 //   output alu_n;           //most significant bit of out, true when Out is negative

      reg [31:0] out = 0;

      wire [31:0] shift_right_out; //alufn controls difference b/n SR and SRA
      shift_rightc X_sr(alufn[1], a, b[4:0], shift_right_out);

      always @ * begin

            out = 32'h0;

            case(alufn)

                  //ADD
                  6'b000_000: begin out = a + b;
                  end

                  //SUB
                  6'b000_001: begin out = a - b;
                  end

                  //MUL
                  6'b000_010: begin out = a * b;
                  end

                  //DIV
                  //6'b000_100: begin out = a / b; COMPILE ERROR
                  //end

                  //AND
                  6'b011_000: begin out = a & b;
                  end
```

```
            //OR
            6'b011_110: begin out = a | b;
            end

            //XOR
            6'b010_110: begin out = a ^ b;
            end

            //LDR "A"
            6'b011_010: begin out = a;
            end

            //SHL
            6'b100_000: begin out = a << b;
            end

            //SHR//out = a >> b;
            6'b100_001: begin out = shift_right_out;
            end

            //SRA
            6'b100_011: begin out = shift_right_out;
            end

            //CMPEQ
            6'b110_011: begin out = (a == b)  ? 32'b1 : 32'b0;
            end

            //CMPLT
            6'b110_101: begin out = (a < b) ? 32'b1 : 32'b0;
            end

            //CMPLE
            6'b110_111: begin out = (a <= b)  ? 32'b1 : 32'b0;
            end

            default: begin out = 32'h0;
            end

        endcase

    end

endmodule
```

Private_Decoder.v

```
////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    15:16:45 05/12/2007
// Design Name:
// Module Name:    ram_decoder
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////
module ram_decoder(clk, beta_ma, beta_mwd, beta_mrd, beta_moe, beta_we
```

```
                                                        , en_ram, ram_ma, ram_in, ram_out
                                                        , moe, we, ma, mwd, mdin
                                                        , gol_sel_out, b_s_data, en_g_b_s);
      // system clock
      input clk;

      // signals from/to beta
      input [31:0] beta_ma;
      input [31:0] beta_mwd;
      output [31:0] beta_mrd;
      input beta_moe;
      input beta_we;

      // from/to main memory
      output en_ram;
      output [31:0] ram_ma;
      output [31:0] ram_in;
      input [31:0] ram_out;

      // to arbiter
      output moe;
      output we;
      output [31:0] ma;
      output [31:0] mwd;

      // from shared data
      input [31:0] mdin;
      input gol_sel_out;
      output b_s_data;
      output en_g_b_s;

      // for now, we're just passing through this signals through from the Beta
      assign ma = beta_ma;
      assign ram_ma = beta_ma;
      assign mwd = beta_mwd;
      assign ram_in = beta_mwd;

      assign b_s_data = beta_mwd[0];

      wire highmem = &beta_ma[31:15]; //denotes if we are using "non-main memory"
resources
      assign sel_ram = !highmem;    // denotes that we're dealing with RAM
      wire gol_buffer_sel     = highmem && (beta_ma[15:0]==16'hff00); // dealing with
the buffer select register

      assign en_ram = sel_ram & beta_we;  // enable write to RAM
      assign en_g_b_s = gol_buffer_sel & beta_we;
      assign moe = beta_moe & ~sel_ram & ~gol_buffer_sel; // pass through moe if we're
not dealing with RAM or buffersel
      assign we = beta_we & ~sel_ram & ~gol_buffer_sel; // pass through we if we're not
dealing with RAM or buffersel

      // selector bits for Beta read MUX, the Beta only reads from RAM, char buffer,
GOL buffer, or keyboard
   reg [1:0] mrd_sel;
   always @ (posedge clk) begin
         // mrd_sel = 2 for RAM, 1 for gol buffer select register, 0 for shared
         mrd_sel <= sel_ram ? 2 : gol_buffer_sel ? 1 : 0;
   end

   // MUX to pick which data to send back to Beta
      //assign beta_mrd = mrd_sel ? ram_out : gol_buffer_sel ? {31'b0,gol_sel_out} :
mdin;

//    assign beta_mrd = sel_ram ? ram_out :
      //                                gol_buffer_sel ? {31'b0,gol_sel_out} :
      //                                mdin;
```

```
	reg [31:0] beta_mrd;

	always @ * begin
		case(mrd_sel)
			2'd2: beta_mrd = ram_out;
			2'd1: beta_mrd = {31'b0, gol_sel_out};
			default: beta_mrd = mdin;
		endcase
	end


endmodule
```

Software.v

```
// single-port read/write memory initialized with Beta_OS_203f0 code

module Beta_OS_203f0test(addr,clk,din,dout,we);
  input [13:0] addr;     // up to 16K locations
  input clk;             // memory has internal address regs
  input [31:0] din;      // appears after rising clock edge
  output [31:0] dout;    // written at rising clock edge
  input we;              // enables write port

  // we're using 2138 out of 4096 locations
  RAMB16_S4
m1test(.CLK(clk),.ADDR(addr[11:0]),.DI(din[3:0]),.DO(dout[3:0]),.WE(we),.EN(1'b1),.SSR(
1'b0));

// Note that more RAM16_S4 instantiations are generated from Python script as well as
// many many many defparam statements.

Endmodule
```

Arbiter.v

```
////////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer: CHRISTOPHER CELIO
//
// Create Date:    17:36:58 05/11/2007
// Design Name:
// Module Name:    memory_arbiter
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////////
module memory_arbiter_quadcore(clk, reset, debug, core_count,

	ma0, ma1, ma2, ma3,

	moe,

	mwd0, mwd1, mwd2, mwd3,

	wr,

	stall,
```

```
   ma_out,

   mwd_out, wr_out, moe_out);

 input clk;
 input reset;
 input debug;        //if true, only let CPU#0 get access
 input [7:0] core_count;
 input [31:0] ma0; //memory address
    input [31:0] ma1; //memory address
    input [31:0] ma2; //memory address
    input [31:0] ma3; //memory address
 input [3:0] moe;
 input [31:0] mwd0;  //memory write data
    input [31:0] mwd1;     //memory write data
    input [31:0] mwd2;     //memory write data
    input [31:0] mwd3;     //memory write data
 input [3:0] wr;
 output [3:0] stall;
    output [31:0] ma_out;
    output [31:0] mwd_out;
    output wr_out;
    output moe_out;

   parameter NCORES = 4;
   //BUG how to deal if second core is *already* accessing memory....

   // wire idle = ~((|moe) | (|wr));
   //reg collision;
   wire [(NCORES-1):0] accessors = moe | wr;
   //wire [1:0] mask = (assessors==2'b11) ? 2'b01 : 2'b00;
   //assign stall = assessors & mask;


   //wire [(NCORES-1):0] mask = 4'hF;

   reg [(NCORES-1):0] stall;
   reg[3:0] mwdsel_temp;
   wire[3:0] mwdsel;

   //Connect the winning Beta to the Memory Bus
   //AGGHH. arrays are as follows:
   // {wr_n, wr_(n-1)...., wr2, wr1, wr0}
   //therefore, wr[n] is for wr_n,
   //but mwdsel== n, is for ma0.
   //this is okay for Debug with CORE#0
   //reg prev_moe1;
   ///reg only1;

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

   //parameter SINGLE_CYCLE = 3'd0;
   //parameter RD0 = 3'd1;
   //parameter RD1 = 3'd2;

   assign mwdsel = debug ? 4'h0 : mwdsel_temp;

   reg [31:0] ma_out;
   reg [31:0] mwd_out;
   reg wr_out, moe_out;

   always @ * begin

   ma_out = ma0;
   mwd_out = mwd0;
   wr_out = wr[0];
```

```
moe_out = moe[0];


      case(mwdsel)

      4'd0:       begin
                        ma_out = ma0;
                        mwd_out = mwd0;
                        wr_out = wr[0];
                        moe_out = moe[0];
                        end
      4'd1:       begin
                        ma_out = ma1;
                        mwd_out = mwd1;
                        wr_out = wr[1];
                        moe_out = moe[1];
                        end
      4'd2:       begin
                        ma_out = ma2;
                        mwd_out = mwd2;
                        wr_out = wr[2];
                        moe_out = moe[2];
                        end
      4'd3:       begin
                        ma_out = ma3;
                        mwd_out = mwd3;
                        wr_out = wr[3];
                        moe_out = moe[3];
                        end
      default:    begin
                        ma_out = ma0;
                        mwd_out = mwd0;
                        wr_out = wr[0];
                        moe_out = moe[0];
                        end
      endcase

end

//Figure out (ie, hack) which Beta gets access to the Memory Bus
//below, is a craptastic method that basically says
//core#0 ALWAYS gets it over core#1, which of course,
//only works for 2-beta systems....

always @ * begin

            stall = {4'b0000};
            mwdsel_temp = 4'd0;

      if (accessors[0]) begin
            stall = {accessors[3:1], 1'b0};
            mwdsel_temp = 4'd0;
      end
      else if (accessors[1]) begin
            stall = {accessors[3:2], 2'b00};
            mwdsel_temp = 4'd1;
      end
      else if (accessors[2]) begin
            stall = {accessors[3], 3'b000};
            mwdsel_temp = 4'd2;
      end
      else if (accessors[3]) begin
            stall = {4'b0000};
            mwdsel_temp = 4'd3;
      end
      else begin
            stall = {4'b0000};
```

```
                mwdsel_temp = 4'd0;
            end
        end

        /*
        integer i, count;

        //detect collision by counting the number of bits on wr, moe busses
        always @ (posedge clk) begin

            count = 0; //count the number of cores trying to access memory
            collision;
            for(i=0; i < NCORES; i = i + 1) begin

                if(moe[i]) count = count + 1;
                if(wr[i]) count = count + 1;

            end

            if(count > 1) collision = 1;

            for(i=0; i <NCORES; i = i + 1) begin


            end
        end
        */

Endmodule
```

Shared_Memory_Decoder.v

```
module mem_decoder(clk, mwe, moe, maddr, mdin,
            en_ram, rd_ps2, en_char, en_gol, en_g_b_s, en_sync,
            sel_ram, sel_char, sel_timer, sel_ps2, sel_gol_a, sel_gol_b, sel_sync,
            //gol_buffer_sel,
            ram_out, ps2_out, char_out, gol_out, sync_out, golsel_out);

        // System clock
        input clk;

        // Beta signals
        input mwe;
        input moe;
        input [31:0] maddr;
        output [31:0] mdin;

        // enable bits
        output en_ram, rd_ps2, en_char, en_gol, en_g_b_s, en_sync;
        output sel_ram, sel_char, sel_timer, sel_ps2, sel_gol_a, sel_gol_b, sel_sync;
        //output gol_buffer_sel;

        // data read from the various memories
        input [31:0] ram_out, ps2_out, char_out, gol_out;
        input golsel_out, sync_out;

        //    General Purpose Private RAM
        //    "60Hz" Clock register
        //    CHAR buffer
        //    GAME Buffer #1
        //    GAME Buffer #2
        //    Display Controller Register(s)
        // Shared RAM (only a small array of booleans for synchronizing purposes)

        // decode memory address
        wire highmem = &maddr[31:15]; //denotes if we are using "non-main memory"
resources
        assign sel_ram          = !highmem;
```

```
      assign sel_char         = highmem && (maddr[15:0] < 16'h8d00);
      assign sel_gol_a        = highmem && (maddr[15:0] < 16'hc600) & ~sel_char;
      assign sel_gol_b        = highmem && (maddr[15:0] < 16'hff00) & ~sel_gol_a &
~sel_char;
      wire gol_buffer_sel     = highmem && (maddr[15:0]==16'hff00);
      assign sel_sync         = highmem && (maddr[15:0] < 16'hFFF8) && (maddr[15:0] >
16'hFF00);
      assign sel_ps2          = highmem && (maddr[15:0]==16'hFFF8);
      assign sel_timer        = highmem && (maddr[15:0]==16'hFFFC);

      assign en_ram = sel_ram & mwe;                                          // enable
write to RAM
      assign en_char = sel_char & mwe;                                        // enable
write to character buffer
      assign en_gol = (sel_gol_a | sel_gol_b) & mwe;         // enable write to GOL
buffers
      assign en_g_b_s = gol_buffer_sel & mwe;                     // enable write to
GOL buffer display bit
      assign en_sync = sel_sync & mwe;

      // selector bits for Beta read MUX, the Beta only reads from RAM, char buffer,
GOL buffer, or keyboard
   reg [2:0] mdin_sel;
      reg rd_ps2;
   always @ (posedge clk) begin
            mdin_sel <= sel_sync ? 5 :
                                    gol_buffer_sel ? 4 :
                                    (sel_gol_a | sel_gol_b) ? 3 :
                                    sel_char ? 2 :
                                    sel_ps2 ? 1 :
                                    0;
            rd_ps2 <= sel_ps2 & !mwe & moe;
   end

   // MUX to pick which data to send back to Beta
      reg [31:0] mdin;
   always @ (mdin_sel or ram_out or ps2_out or char_out or gol_out or golsel_out or
sync_out)
            case (mdin_sel)
                  5 : mdin = {31'b0, sync_out};
                  4 : mdin = {31'b0, golsel_out};
                  3 : mdin = gol_out;
                  2 : mdin = char_out;
                  1 : mdin = ps2_out;
                  default : mdin = ram_out;
            endcase

endmodule
```

Sync_Ram.v

```
// Description: This is the only RAM that needs to be shared between cores
//                          b/c its small, i've made it a seperate, optimized module
//                          This is a "check-in" list. when I beta finishes a task,
//                          it writes a boolean value (1) in its slot to say it has
finished
//                          Core#0's job, once it has finished the process, is to
monitor
//                          the synch_ram to see if everyone has finished, and then
clears
//                          the ram when everyone has finished.
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
```

```
////////////////////////////////////////////////////////////////////////////////
module synch_ram(clk, reset, debug, mem_addr, data_in, data_out, enable, sync_out);
    input clk;
    input reset;
    input debug;
    input [31:0] mem_addr;
    input data_in;
    output data_out;
    input enable;
       output [63:0] sync_out;

      parameter MEM_LOC = 32'hFFFF_FF04;

      wire [31:0] addr = mem_addr - MEM_LOC;

 // (* ram_style = "distributed" *) //trick learned from Terman
                                                                  //since new verilog
is buggy at implicit ram definitions
      reg [63:0] registers; //[31:0];

      assign sync_out = registers;
      assign data_out = registers[addr[5:0]];

      //write port, if WERF (Write Enable Register File) high
      always @(posedge clk) begin
            if(!reset)
                  registers[addr[5:0]] <= (enable) ? data_in : registers[addr[5:0]];
            else begin
                  //BUG this should be taken out if the beta works
                  registers[63:0] <= 6'b0;
            end
      end
endmodule
```

VGA_TOP.v

```
////////////////////////////////////////////////////////////////////////////////
// Engineer: Jonathan M Long
//
// Module Name: vga_timer
// Description: generates the correct signals needed to drive a VGA displayer
////////////////////////////////////////////////////////////////////////////////
module vga_m(pix_clk, reset, hsout, vsout, blank_b, color, mode
            , cellnum, cell_state, char_addr, char_code, addrb, sel_gol, mit_logo
            //, font_byte, font_addr);
            );
  input pix_clk; // this clock should be twice the needed pixel clock frequency, 63Mhz
  input reset;
  output hsout;
  output vsout;
  output blank_b;
  output [23:0] color;
  input mode;

  output [13:0] cellnum;
  input cell_state;
  output [11:0] char_addr;
  input [7:0] char_code;
  input [13:0] addrb;
  input sel_gol;
  input mit_logo;

  //from font.v
 // output [10:0] font_addr;
 // input [7:0] font_byte;

  // Give Beta access to the character buffer
  //input sys_clock;  //use the system clock for the Beta's read/write char buffer port
```

```
//input mwe;
//input [9:0] maddr;
//input [31:0] mdin;
// output [31:0] mdout;

    // internal wires
    wire hsync, vsync;
    wire [9:0] pixel_count, line_count;
    wire [23:0] gol_color, terminal_color;

vga_controller vgac(  .pixel_clock(pix_clk),
                                                .reset_sync(reset),
                                                .hsync(hsync), .vsync(vsync),
                                                .blank_b(blank_b),
                                                .pixel_count(pixel_count),
                                                .line_count(line_count)
                      );

    sync_delay s_d1(  .pixel_clock(pix_clk),
                                          .hsync(hsync), .vsync(vsync),
                                          .hsync_delayed(hsout),
                                          .vsync_delayed(vsout)
                );

    goldisp gd1(      .vid_clk(pix_clk),
                                  .hcount(pixel_count),
                                  .vcount(line_count),
                                  .color(gol_color),
                                  .cellnum(cellnum),
                                  .cell_state(cell_state),
                                  .addrb(addrb),
                                  .sel_gol(sel_gol),
                                  .show(1'b0)
                      );

    terminaldisp td1( .clk(pix_clk),
                                          .hcount(pixel_count),
                                          .vcount(line_count),
                                          .color(terminal_color),
                                          .char_addr(char_addr),
                                          .char_code(char_code)
                                          //,.font_byte(font_byte)
                                          //,.font_addr(font_addr)
                      );

                      //**************************************
    // MIT Logo as a background option....
    //added by CCelio for "effect"
    //modified from my 6.111 Lab4 Pong code
    //******************************

    wire [23:0] L1,L2,L3,L4,L5,L6,L7;

    //******************************
    //COLOR PARAMETERS

    parameter MIT_CRIMSON   = 24'b0101_1111__0001_1111__0001_1111;
    parameter MIT_GREY            = 24'b0100_1111__0100_1111__0011_1111;
    parameter BLACK               = 24'h000000;
    parameter WHITE               = 24'hFFFFFF;
    parameter BLUE                = 24'h0000FF;

    //pixel parameters
    parameter logo_width          = 39;
    parameter logo_height   = 195;
    parameter logo_start_x  = 65;
    parameter logo_start_y  = 134;
```

```
//MIT LOGO
//the 'M'
 lab4_v_rectangle v_logo_m_1(pix_clk, pixel_count, line_count
                                        ,
logo_start_x,logo_start_y,logo_start_x+logo_width,logo_start_y+logo_height,
MIT_CRIMSON, vsync, L1);
 lab4_v_rectangle v_logo_m_3(pix_clk, pixel_count, line_count
                                        , logo_start_x + 197 -
133,logo_start_y,logo_start_x -133 + 197+logo_width,logo_start_y+137, MIT_GREY, vsync,
L2);
 lab4_v_rectangle v_logo_m_4(pix_clk, pixel_count, line_count
                                        , logo_start_x + 261 -
133,logo_start_y,261+logo_width + logo_start_x - 133,logo_start_y+logo_height,
MIT_CRIMSON, vsync, L3);

 //the 'i'
 lab4_v_rectangle logo_i_1(pix_clk, pixel_count, line_count
                                        , logo_start_x -133 +
325,logo_start_y,logo_start_x -133 + 325+logo_width,logo_start_y+logo_width,
MIT_CRIMSON, vsync, L4);
 lab4_v_rectangle logo_i_2(pix_clk, pixel_count, line_count
                                        , logo_start_x -133 + 325,logo_start_y
+ 2 * logo_width,logo_start_x -133 + 325+logo_width,logo_start_y+logo_height, MIT_GREY,
vsync, L5);

 //the 'T'
 lab4_v_rectangle logo_t_1(pix_clk, pixel_count, line_count
                                        ,logo_start_x -133 +
389,logo_start_y,logo_start_x -133 + 389+117,logo_start_y+logo_width, MIT_CRIMSON,
vsync, L6);
 lab4_v_rectangle logo_t_2(pix_clk, pixel_count, line_count
                                        ,logo_start_x -133 + 389,logo_start_y +
2 * logo_width,logo_start_x -133 + 389+logo_width,logo_start_y+logo_height,
MIT_CRIMSON, vsync, L7);

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

wire [23:0] logo_color = (L1 | L2 | L3 | L4 | L5 | L6 | L7);

reg [23:0] color;
always @(posedge pix_clk)
 color <= mode ? ((gol_color==32'b0) ? (mit_logo ? logo_color : 0 )
                                                              : gol_color)
                                  :

                                  (mit_logo & (terminal_color==32'b0 ||
terminal_color==BLUE)) //ie, background color
                                                    ?     ((logo_color!=32'b0)
? logo_color : terminal_color)
                                                    :
                                                    terminal_color;

//    reg [23:0] color;
//    always @(posedge pix_clk)
//          color <= mode ? gol_color : terminal_color;

Endmodule
```

VGA_Controller.v

```
////////////////////////////////////////////////////////////////////////////////
// Engineer: Jonathan M Long
//
// Module Name: vga_timer
// Description: generates the correct signals needed to drive a VGA displayer
////////////////////////////////////////////////////////////////////////////////
```

```
module vga_controller(pixel_clock, reset_sync, hsync, vsync, blank_b, pixel_count,
line_count);
      input pixel_clock;
      input reset_sync;
      output hsync;
      output vsync;
      output blank_b;
      output [9:0] pixel_count;
      output [9:0] line_count;

      // Default paramter values for a 640 X 480 display
      parameter PIXELS = 800;
      parameter LINES = 525;
      parameter HVID = 640;
      parameter HFRONT = 16;
      parameter HSYNC = 96;
      parameter VVID = 480;
      parameter VFRONT = 11;
      parameter VSYNC = 2;

      //    parameter PIXELS = 1056;
      //    parameter LINES = 624;
      //    parameter HVID = 800;
      //    parameter HFRONT = 16;
      //    parameter HSYNC = 80;
      //    parameter VVID = 600;
      //    parameter VFRONT = 1;
      //    parameter VSYNC = 2;

      reg [9:0] pixel_count = 10'b0;
      reg [9:0] line_count = 10'b0;
      reg hsync = 1'b1;
      reg vsync = 1'b1;
      reg blank_b = 1'b1;

      wire [9:0] next_pix;
      wire [9:0] next_line;

      always @ (posedge pixel_clock)
      begin

            if (reset_sync)
            begin
                  // We're resetting, set all default values
                  hsync <= 1'b1;
                  vsync <= 1'b1;
                  blank_b <= 1'b1;
                  pixel_count <= 10'b0;
                  line_count <= 10'b0;
            end
            else
            begin
                  // hsync is high if the pixel count is outside of the "hsync" segment
                  hsync <= (next_pix < HVID + HFRONT) | (next_pix >= HVID + HFRONT +
HSYNC);

                  // vsync is high if the line count is outside of the "vsync" segment
                  vsync <= (next_line < VVID + VFRONT) | (next_line >= VVID + VFRONT +
VSYNC);

                  // blank_b is high if the pixel and line count are within their
actual display segments
                  blank_b <= (next_pix < HVID) & (next_line < VVID);

                  // increment pixel and line counts
                  pixel_count <= next_pix;
                  line_count <= next_line;
```

```
		end
	end

	// reset pixel count to zero if we've reached the total pixel values, increment
if not there yet
	assign next_pix = (pixel_count == (PIXELS - 1)) ? 10'b0 : pixel_count + 1;

	// reset line count if we've reach the end of the total pixel and line values,
increment if not there yet
	assign next_line = (pixel_count == (PIXELS - 1)) ? (line_count == (LINES - 1)) ?
10'b0 : line_count + 1 : line_count;

endmodule
```

GOL_Disp.v

```
module goldisp(vid_clk, hcount, vcount, color, cellnum, cell_state, addrb, sel_gol,
show);
	input vid_clk;
	input [9:0] hcount;
	input [9:0] vcount;
	output [23:0] color;

	output [13:0] cellnum;
	input cell_state;
	input [13:0] addrb;
	input sel_gol;
	input show;

	// Give Beta access to the game state buffer
	//input sys_clock;  //use the system clock for the Beta's read/write char buffer
port
	//input en_gol;
	//input [9:0] maddr;
	//input [31:0] mdin;
	//output [31:0] mdout;
	//input sel_gol_a, sel_gol_b, en_g_b_s;

	// hreset denotes whenever the we're about to start the next row of pixels
	// needed (800 - 3) instead of the expected (800 - 1) for proper display
	wire hreset = (hcount == (800 - 3));

	// vreset denotes whenever we're about to start top row again.
	wire vreset = hreset & (vcount == (525 - 1));

	parameter BLUE = 24'h0000FF;
	parameter YELLOW = 24'hFFFF00;
	parameter GRAY = 24'hC0C0C0;
	parameter WHITE = 24'hFFFFFF;
	parameter BLACK = 24'h000000;

	reg [7:0] line;       // 0 .. 119, which line of cells are we on?
	reg [7:0] column;    // 0 .. 119, which cell of the line are we on?
	reg [1:0] v;         // 0 .. 3, which row of the cell are we on?
	reg [1:0] h;         // 0 .. 3,  which pixel of the row are we on?

	// Calculate the next line, reset to zero when hreset and vreset are true,
otherwise
	// just increment whenever v==3, else remain the same
	wire [7:0] next_line = hreset ? (vreset ? 8'b0 : (v == 3) ? line + 1 : line) :
line;

	// Calculate the next column, reset to zero when hreset is true, otherwise,
increment whenever
	// h==3, else, remain the same
	wire [7:0] next_column = hreset ? 8'b0 : h == 3 ? column + 1 : column;

	// Update position data every pixel clock cycle
```

```
	always @ (posedge vid_clk) begin
		h <= hreset ? 2'b0 : h+1;
		v <= hreset ? ((vreset || v == 3) ? 2'b0 : v+1) : v;
		column <= next_column;
		line <= next_line;
	end

	//Lookup next cell so there's time to access memory
	assign cellnum = (next_line * 120) + next_column;

	wire [13:0] current_cell = (line * 120) + column;

	wire touched = sel_gol & (current_cell == addrb) & show;

	// Calculate what color to show
	reg [23:0] color;
	always @(posedge vid_clk)
		//	beyond the game board, draw a white wall at right edge
		color <= (column >= 122) ? BLACK : (column >= 120) ? WHITE : touched ? BLUE
: cell_state ? WHITE : BLACK; //YELLOW : GRAY;

endmodule
```

Terminal_Display.v

```
module terminaldisp(clk, hcount, vcount, color, char_addr, char_code);//,
font_addr,font_byte);
	input clk;
	input [9:0] hcount;
	input [9:0] vcount;
	output [23:0] color;

	output [11:0] char_addr;
	input [7:0] char_code;


	//out to font module
	//output [10:0] font_addr;
	//input [7:0] font_byte;

	// Some colors
	parameter BLACK = 24'h000000;
	parameter WHITE = 24'hFFFFFF;
	parameter RED = 24'hFF0000;
	parameter GREEN = 24'h00FF00;
	parameter BLUE = 24'h0000FF;

	wire hreset = (hcount == (800 - 3));
	wire vreset = hreset & (vcount == (525 - 1));

	reg [6:0] char;		// 0 .. 79, which char of the line are we on?
	reg [5:0] line;		// 0 .. 39, which line of chars are we on?
	reg [3:0] v;		// 0 .. 11, which row of the char are we on?
	reg [2:0] h;		// 0 .. 7,  which pixel of the row are we on?

	wire [6:0] next_char = hreset ? 7'b0 : h == 7 ? char + 1 : char;
	wire [5:0] next_line = hreset ? (vreset ? 6'b0 : (v == 11) ? line + 1 : line) :
line;

	always @ (posedge clk) begin
		h <= hreset ? 3'b0 : (h + 1);
		v <= hreset ? ((vreset || v == 11) ? 4'b0 : v + 1) : v;
		char <= next_char;
		line <= next_line;
	end

	//Lookup next row and col so there's time to process result
	assign char_addr = (next_line * 80) + next_char;
```

```
      // The Character Buffer, use the inverse pixel clock to allow time to retrieve
character code
      //cmem b(.addra(char_addr), .clka(~clk), .douta(char_code), .addrb(maddr),
.clkb(sys_clock), .dinb(mdin), .doutb(mdout), .web(mwe));



      reg reverse;      //REVERSES the color of the line if MSB of buffer entry is HIGH
      always @ (posedge clk) reverse <= char_code[7];
      wire [10:0] font_addr = char_code[6:0] * 12 + v;
      wire [7:0] font_byte; //Holds one row from the FONT ROM

      font f(.addr(font_addr), .clk(clk), .row(font_byte));  //The FONT ROM

      reg [23:0] color;
      always @(posedge clk)
            color <= (font_byte[7 - h] ^ reverse) ? WHITE: BLUE; //GREEN : BLACK;

endmodule
```

Font_ROM.v

```
// 8x12 font memory for 128 chars
module font(addr,clk,row);
  input clk;
  input [10:0] addr;
  output [7:0] row;

  // font read-only memory: (128 * 12row/chars) x (8 bits/row)
  RAMB16_S9
  f (.CLK(clk),.ADDR(addr),.DO(row),
              .WE(1'b0),.EN(1'b1),.SSR(1'b0));

// Note that this module will be followed by defparam statements when generated from
python script

Endmodule
```

Character_RAM.v

```
// build an 80x24 character memory with two ports:
// * one is 8 bits wide (and read-only) for use by the video refresh circuitry
// * one is 32 bits wide for use by the CPU
module cmem(addra,clka,douta,addrb,clkb,dinb,doutb,web);
   input [11:0] addra;
   input clka;
   output [7:0] douta;
   input [9:0] addrb;
   input clkb;
   input [31:0] dinb;
   output [31:0] doutb;
   input web;

   // use 2 BRAMs
   // port A: 4K x 4
   // port B: 1K x 16
   RAMB16_S4_S18 mlo(.CLKA(clka),.ADDRA(addra),.DOA(douta[3:0]),
                     .WEA(1'b0),.ENA(1'b1),.SSRA(1'b0),
                     .CLKB(clkb),.ADDRB(addrb),
                     .DIB({dinb[27:24],dinb[19:16],dinb[11:8],dinb[3:0]}),
                 .DIPB(2'd0),
                     .DOB({doutb[27:24],doutb[19:16],doutb[11:8],doutb[3:0]}),
                     .WEB(web),.ENB(1'b1),.SSRB(1'b0)
                     ),
                 mhi(.CLKA(clka),.ADDRA(addra),.DOA(douta[7:4]),
                 .WEA(1'b0),.ENA(1'b1),.SSRA(1'b0),
                     .CLKB(clkb),.ADDRB(addrb),
```

```
                    .DIB({dinb[31:28],dinb[23:20],dinb[15:12],dinb[7:4]}),
                .DIPB(2'd0),
                    .DOB({doutb[31:28],doutb[23:20],doutb[15:12],doutb[7:4]}),
                    .WEB(web),.ENB(1'b1),.SSRB(1'b0)
                 );
Endmodule
```

GOL_Mem_Manager.v

```
module gol_mem_manager(addra, clka, douta, addrb, clkb, dinb, doutb, web,
                              sel_gol_a, sel_gol_b, en_g_b_s, g_b_s_wdata,
                              golram_select, debug, reset, busy, rom_sel);

      // go to GoL buffer
      input [13:0] addra;
      input clka;
      output douta;
      input [13:0] addrb;
      input clkb;
      input dinb;
      output [31:0] doutb;
      input web;
      input sel_gol_a, sel_gol_b;
      input en_g_b_s;
      input g_b_s_wdata;
      output golram_select;
      input debug;

      input reset; // used to load from ROM, happens on nededge
      output busy; // stalls Betas while RAM is loaded
      input rom_sel;

      wire [13:0] rom_addr, ram_addr;
      wire load_we, busy, start;

      gol_load_engine gle1(clkb, start, rom_addr, ram_addr, load_we, busy, reset);

      level_to_pulse ltp1(clkb, reset, start);

      wire rom_out0, rom_out1;
      gol_romA gr0(rom_addr, clkb, rom_out0);
      gol_romB gr1(rom_addr, clkb, rom_out1);

      wire rom_out = rom_sel ? rom_out1 : rom_out0;

      golmem gm1( .addra(addra), .clka(clka), .douta(douta),
                              .addrb(busy ? ram_addr : addrb), .clkb(clkb), .dinb(busy
? rom_out : dinb), .doutb(doutb), .web(busy ? load_we : web),
                              .sel_gol_a(busy ? 1 : sel_gol_a), .sel_gol_b(busy ? 0 :
sel_gol_b)
                              , .en_g_b_s(busy ? 0 : en_g_b_s)
                              , .golram_select(golram_select),
.g_b_s_wdata(g_b_s_wdata)
                              ,.debug(debug)
                 );


Endmodule
```

GoL_Load_Engine.v

```
module gol_load_engine(clk, start, rom_addr, ram_addr, write_enable, busy, reset);
    input clk;
       input start;
    output [13:0] rom_addr;
    output [13:0] ram_addr;
       output write_enable;
       output busy;
```

```
    input reset;

       parameter NUM_LOCATIONS = 14400;

       reg [13:0] rom_addr;
       reg write_enable;
       reg busy;
       reg [13:0] count;

       assign ram_addr = rom_addr - 1;

       always @ (posedge clk) begin
            if (reset) begin
                  rom_addr <= 14'h0;
                  write_enable <= 0;
                  count <= 14'h0;
                  busy <= 0;
            end

            else if (start) begin
                  rom_addr <= 14'h0;
                  write_enable <= 0;
                  count <= 14'h0;
                  busy <= 1;
            end

            else if (busy) begin
                  if (rom_addr == NUM_LOCATIONS) begin //We're done
                        busy <= 0; // stop engine
                        write_enable <= 0; // stop writing to ram
                        busy <= 0;
                  end
                  else begin // we're not done
                        rom_addr <= rom_addr + 1; // increment rom address
                        write_enable <= 1; // keep the RAM enabled to write
                  end
            end

       end
endmodule
```

Level_To_Pulse.v

```
module level_to_pulse(clk, signal, out);
      input clk;
      input signal;
      output out;

      parameter LOW = 0;
      parameter HIGH = 1;

      reg state = 0, next_state;
      reg out, next_out;

      always @ (posedge clk) begin
            state <= next_state;
            out <= next_out;
      end

      always @ * begin
            next_out = 0;

            case(state)

            LOW:        begin
                              if (signal) next_state = HIGH;
                              else  next_state = LOW;
                              end
```

```
        HIGH:       begin
                        if (!signal) begin
                              next_state = LOW;
                              next_out = 1;
                        end
                        else next_state = HIGH;
                        end
        endcase
    end
endmodule
```

GoL_ROM.v

```
module gol_romA(addr, clk, dout);
    input [13:0] addr;
    input clk;
    output dout;

    RAMB16_S1 romA(.CLK(clk),.ADDR(addr),.DO(dout),
                                        .WE(1'b0),.EN(1'b1),.SSR(1'b0));

// defparam statements go here when generated from script

Endmodule
```

GoL_RAM.v

```
module golmem(addra, clka, douta, addrb, clkb, dinb, doutb, web, sel_gol_a, sel_gol_b,
en_g_b_s
                        , g_b_s_wdata , golram_select, debug);
    input [13:0] addra;
    input clka;
    output douta;
    input [13:0] addrb;
    input clkb;
    input dinb;
    output [31:0] doutb;
    input web;
      input sel_gol_a, sel_gol_b;
      input en_g_b_s;
      input g_b_s_wdata;
      output golram_select;
      input debug;

      // update the buffer display register if enable_GOL_buffer_sel is true
      // if lowest bit of the Beta's word is LOW, specifies gol_a
      // if lowest bit of the Beta's word is HIGH,  specifies gol_b
      reg ram_select = 0;
      always @ (posedge clkb)
           if (en_g_b_s) ram_select <= g_b_s_wdata;
           //ram_select <= debug;

     assign golram_select = ram_select;

      // setup write enable for the two buffers
      wire we_a = sel_gol_a & web;
      wire we_b = sel_gol_b & web;

      wire gol_a_out, gol_b_out, lo_out, hi_out;

      // Multiplex output to Display Controller
      assign douta = ram_select ? gol_b_out : gol_a_out;

      // Multiplex output to Beta(s), pad with zeros
      assign doutb = sel_gol_a ? {31'b0, lo_out} : {31'b0, hi_out};
```

```
       RAMB16_S1_S1 bufferA(.CLKA(clka),.ADDRA(addra[13:0]),.DOA(gol_a_out),
                    .WEA(1'b0),.ENA(1'b1),.SSRA(1'b0),
                    .CLKB(clkb),.ADDRB(addrb[13:0]),
                    .DIB(dinb),
                                         //.DIPB(2'd0),
                    .DOB(lo_out),
                    .WEB(we_a),.ENB(1'b1),.SSRB(1'b0)
                    ),

      bufferB(.CLKA(clka),.ADDRA(addra[13:0]),.DOA(gol_b_out),
                    .WEA(1'b0),.ENA(1'b1),.SSRA(1'b0),
                    .CLKB(clkb),.ADDRB(addrb[13:0]),
                    .DIB(dinb),
                                         //.DIPB(2'd0),
                    .DOB(hi_out),
                    .WEB(we_b),.ENB(1'b1),.SSRB(1'b0)
                    );

Endmodule
```

PS2.v

```
//modified by CCelio and MLong for multi-core capabilities

module ps2(clk,reset,watchdog,ps2c,ps2d,fifo_rd,fifo_data,fifo_empty,fifo_overflow,
core_count);
  input clk,reset,watchdog,ps2c,ps2d;
  input fifo_rd;
  output [7:0] fifo_data;
  output fifo_empty;
  output fifo_overflow;
  input signed [8:0] core_count;

  reg [3:0] count;      // count incoming data bits
  reg [9:0] shift;      // accumulate incoming data bits

  reg [7:0] fifo[7:0];   // 8 element data fifo
  reg fifo_overflow;
  reg [2:0] wptr,rptr;   // fifo write and read pointers
  reg [7:0] rd_count;   // added by M. Long, count how many cores have accessed
                                                // the FIFO buffer. allow all cores to
read the PS2 fifo
                                                // before incrementing the read_pointer
(rptr).

  wire [2:0] wptr_inc = wptr + 1;

  assign fifo_empty = (wptr == rptr);
  assign fifo_data = fifo[rptr];

  // synchronize PS2 clock to local clock and look for falling edge
  reg [2:0] ps2c_sync;
  always @ (posedge clk) ps2c_sync <= {ps2c_sync[1:0],ps2c};
  wire sample = ps2c_sync[2] & ~ps2c_sync[1];

  reg timeout;
  always @ (posedge clk) begin
    if (reset) begin
      count <= 0;
      wptr <= 0;
      rptr <= 0;
      timeout <= 0;
      fifo_overflow <= 0;
    end else if (sample) begin
      // order of arrival: 0,8 bits of data (LSB first),odd parity,1
      if (count==10) begin
        // just received what should be the stop bit
        if (shift[0]==0 && ps2d==1 && (^shift[9:1])==1) begin
```

```
          fifo[wptr] <= shift[8:1];
          wptr <= wptr_inc;
        fifo_overflow <= fifo_overflow | (wptr_inc == rptr);
        end
        count <= 0;
        timeout <= 0;
      end else begin
        shift <= {ps2d,shift[9:1]};
        count <= count + 1;
      end
    end else if (watchdog && count!=0) begin
      if (timeout) begin
        // second tick of watchdog while trying to read PS2 data
      count <= 0;
        timeout <= 0;
      end else timeout <= 1;
    end

    // bump read pointer if we're done with current value.
    // Read also resets the overflow indicator
    if (fifo_rd && !fifo_empty) begin
   //   rptr <= rptr + 1;
    //  fifo_overflow <= 0;

      // if(fifo_rd) rd_count <= rd_count + 1; //POSSIBLE BUG: how often is this
incremented?
                                                //problem is that RD's are two cycles
                                                //down below, this doesn't get
incremented until the NEXT turn
                                                //optimize + make it core_count-1 &&
fifo_rd....
      // if (rd_count >= (core_count-1) && fifo_rd && !fifo_empty) begin
            rptr <= rptr + 1;
            fifo_overflow <= 0;
            rd_count <=0;
    end

  end
endmodule
```

Hex_Display.v

```
module display_16hex (reset, clock_27mhz, data,
            disp_blank, disp_clock, disp_rs, disp_ce_b,
            disp_reset_b, disp_data_out);

   input reset, clock_27mhz;    // clock and reset (active high reset)
   input [63:0] data;         // 16 hex nibbles to display

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

   reg disp_data_out, disp_rs, disp_ce_b, disp_reset_b;

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

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

   always @(posedge clock_27mhz)
```

```
  begin
   if (reset)
     begin
        count = 0;
        clock = 0;
     end
   else if (count == 26)
     begin
        clock = ~clock;
        count = 5'h00;
     end
   else
     count = count+1;
  end

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

assign dreset = (reset_count != 0);

assign disp_clock = ~clock;

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

reg [7:0] state;           // FSM state
reg [9:0] dot_index;       // index to current dot being clocked out
reg [31:0] control;        // control register
reg [3:0] char_index;      // index of current character
reg [39:0] dots;           // dots for a single digit
reg [3:0] nibble;          // hex nibble of current character

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

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

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

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

    8'h03:
      begin
         // Latch dot data
         disp_ce_b <= 1'b1;
         dot_index <= 31;        // re-purpose to init ctrl reg
         disp_rs <= 1'b1; // Select the control register
         state <= state+1;
      end

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

    8'h05:
      begin
         // Latch the control register data / dot data
         disp_ce_b <= 1'b1;
         dot_index <= 39;        // init for single char
         char_index <= 15;       // start with MS char
         state <= state+1;
         disp_rs <= 1'b0;        // Select the dot register
      end

    8'h06:
      begin
         // Load the user's dot data into the dot reg, char by char
         disp_ce_b <= 1'b0;
         disp_data_out <= dots[dot_index]; // dot data from msb
         if (dot_index == 0)
           if (char_index == 0)
             state <= 5;               // all done, latch data
         else
         begin
           char_index <= char_index - 1;     // goto next char
           dot_index <= 39;
         end
         else
         dot_index <= dot_index-1;     // else loop thru all dots
      end

    endcase

always @ (data or char_index)
  case (char_index)
    4'h0:           nibble <= data[3:0];
    4'h1:           nibble <= data[7:4];
    4'h2:           nibble <= data[11:8];
    4'h3:           nibble <= data[15:12];
```

```
      4'h4:           nibble <= data[19:16];
      4'h5:           nibble <= data[23:20];
      4'h6:           nibble <= data[27:24];
      4'h7:           nibble <= data[31:28];
      4'h8:           nibble <= data[35:32];
      4'h9:           nibble <= data[39:36];
      4'hA:           nibble <= data[43:40];
      4'hB:           nibble <= data[47:44];
      4'hC:           nibble <= data[51:48];
      4'hD:           nibble <= data[55:52];
      4'hE:           nibble <= data[59:56];
      4'hF:           nibble <= data[63:60];
    endcase

  always @(nibble)
    case (nibble)
      4'h0: dots <= 40'b00111110_01010001_01001001_01000101_00111110;
      4'h1: dots <= 40'b00000000_01000010_01111111_01000000_00000000;
      4'h2: dots <= 40'b01100010_01010001_01001001_01001001_01000110;
      4'h3: dots <= 40'b00100010_01000001_01001001_01001001_00110110;
      4'h4: dots <= 40'b00011000_00010100_00010010_01111111_00010000;
      4'h5: dots <= 40'b00100111_01000101_01000101_01000101_00111001;
      4'h6: dots <= 40'b00111100_01001010_01001001_01001001_00110000;
      4'h7: dots <= 40'b00000001_01110001_00001001_00000101_00000011;
      4'h8: dots <= 40'b00110110_01001001_01001001_01001001_00110110;
      4'h9: dots <= 40'b00000110_01001001_01001001_00101001_00011110;
      4'hA: dots <= 40'b01111110_00001001_00001001_00001001_01111110;
      4'hB: dots <= 40'b01111111_01001001_01001001_01001001_00110110;
      4'hC: dots <= 40'b00111110_01000001_01000001_01000001_00100010;
      4'hD: dots <= 40'b01111111_01000001_01000001_01000001_00111110;
      4'hE: dots <= 40'b01111111_01001001_01001001_01001001_01000001;
      4'hF: dots <= 40'b01111111_00001001_00001001_00001001_00000001;
    endcase

endmodule
```

Beta_OS_203.uasm

```
||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
|||  Lab #9: Simple OS demo for 6.004 Beta processor
|||  revision: Christopher Celio Spring 2007, for 6.111
|||          User processes 1 and 2 have been modified for
|||          multi-core demonstration. P2 plays the Game of Life
|||          P1 has CPU#0 echo back user input, and appends the Core's ID
|||          and the total count of cores. All other cores ping in as well
|||
||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||

|||CCelio:  This OS has been taken and used for the 6.111 quad-core demo

|||   This file is a modification of os.uasm, kernel.uasm and user.uasm,
|||   all posted in /mit/6.004/bsim.  The original demo was constructed
|||   in Fall of 1994 by Steve Ward.

|||   This program implements a primitive OS kernel for the Beta
|||   along with three simple user-mode processes hooked together thru
|||   a semaphore-controlled bounded buffer.
|||
|||   The three processes -- and the kernel -- share an address space;
|||   each is allocated its own stack (for a total of 4 stacks), and
|||   each process has its own virtual machine state (ie, registers).
|||   The latter is stored in the kernel ProcTbl, which contains a data
|||   structure for each process.

.include beta.uasm            | Define Beta instructions, etc.
.options clock tty

||| The following code is a primitive but complete timesharing kernel
```

```
|||  sufficient to run three processes, plus handlers for a small
|||  selection of supervisor calls (SVCs) to perform OS services.
|||  The latter include simple console I/O and semaphores.
|||
|||  All kernel code is executed with the Kernel-mode bit of the
|||   program counter -- its high-order bit --- set.  This causes
|||   new interrupt requests to be deferred until the kernel returns
|||   to user mode.

||| Interrupt vectors:

. = VEC_RESET
      BR(I_Reset) | on Reset (start-up)
. = VEC_II
      BR(I_IllOp) | on Illegal Instruction (eg SVC)
. = VEC_CLK
      BR(I_Clk)   | On clock interrupt
. = VEC_KBD
      BR(I_Kbd)   | on Keyboard interrupt
. = VEC_MOUSE
      BR(I_BadInt)      | on mouse interrupt

||| The following macro is the first instruction to be entered for each
||| asynchronous I/O interrupt handler.    It adjusts XP (the interrupted
||| PC) to account for the instruction skipped due to the pipeline bubble.
.macro ENTER_INTERRUPT SUBC(XP,4,XP)

||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
||| Kernel Interrupt support code
||| We use a slightly simpler (and less efficient) scheme here from
|||  that in the text.  On kernel entry, the ENTIRE state -- 31
|||  registers -- of the interrupted program is saved in a designated
|||  region of kernel memory ("UserMState", below).  This entire state
|||  is then restored on return to the interrupted program.
||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||

| Here's the SAVED STATE of the interrupted process, while we're
| processing an interrupt.
UserMState:
      STORAGE(32) | R0-R31... (PC is in XP!)

| Here are macros to SAVE and RESTORE state -- 31 registers -- from
|   the above storage.

| N.B. - The following macro assumes that R0 is a macro for
| the integer 0, R1 is a macro for the integer 1, etc.
.macro SS(R) ST(R, UserMState+(4*R))      | (Auxiliary macro)

.macro SAVESTATE() {
      SS(0)  SS(1)  SS(2)  SS(3)  SS(4)  SS(5)  SS(6)  SS(7)
      SS(8)  SS(9)  SS(10) SS(11) SS(12) SS(13) SS(14) SS(15)
      SS(16) SS(17) SS(18) SS(19) SS(20) SS(21) SS(22) SS(23)
      SS(24) SS(25) SS(26) SS(27) SS(28) SS(29) SS(30) }

| See comment for SS(R), above
.macro RS(R) LD(UserMState+(4*R), R)      | (Auxiliary macro)

.macro RESTORESTATE() {
      RS(0)  RS(1)  RS(2)  RS(3)  RS(4)  RS(5)  RS(6)  RS(7)
      RS(8)  RS(9)  RS(10) RS(11) RS(12) RS(13) RS(14) RS(15)
      RS(16) RS(17) RS(18) RS(19) RS(20) RS(21) RS(22) RS(23)
      RS(24) RS(25) RS(26) RS(27) RS(28) RS(29) RS(30) }

KStack:     LONG(.+4)           | Pointer to ...
      STORAGE(256)              |  ... the kernel stack.

|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
```

```
||| Handler for unexpected interrupts
|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||

I_BadInt:
      CALL(KWrMsg)                   | Type out an error msg,
      .text "Unexpected interrupt..."
      BR(.)

|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
||| Handler for Illegal Instructions
|||  (including SVCs)
|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||

I_IllOp:
      SAVESTATE()       | Save the machine state.
      LD(KStack, SP)          | Install kernel stack pointer.

      LD(XP, -4, r0)          | Fetch the illegal instruction
      SHRC(r0, 26, r0)  | Extract the 6-bit OPCODE
      SHLC(r0, 2, r0)         | Make it a WORD (4-byte) index
      LD(r0, UUOTbl, r0)      | Fetch UUOTbl[OPCODE]
      JMP(r0)                 | and dispatch to the UUO handler.

.macro UUO(ADR) LONG(ADR+PC_SUPERVISOR)   | Auxiliary Macros
.macro BAD()      UUO(UUOError)

UUOTbl:     BAD()       UUO(SVC_UUO)      BAD()       BAD()
      BAD()       BAD()       BAD()       BAD()
      BAD()       BAD()       BAD()       BAD()
      BAD()       BAD()       BAD()       BAD()
      BAD()       BAD()       BAD()       BAD()
      BAD()       BAD()       BAD()       BAD()
      BAD()       BAD()       BAD()       BAD()
      BAD()       BAD()       BAD()       BAD()

||| Here's the handler for truly unused opcodes (not SVCs):
UUOError:
      CALL(KWrMsg)                   | Type out an error msg,
      .text "Illegal instruction "
      LD(xp, -4, r0)                 |   giving hex instr and location;
      CALL(KHexPrt)
      CALL(KWrMsg)
      .text " at location 0x"
      SUBC(xp,-4,r0)
      CALL(KHexPrt)
      CALL(KWrMsg)
      .text "! ....."
      BR(.)                   | Then crash system.

||| Here's the common exit sequence from Kernel interrupt handlers:
||| Restore registers, and jump back to the interrupted user-mode
||| program.

I_Rtn:      RESTORESTATE()
kexit:      JMP(XP)                  | Good place for debugging breakpoint!

||| Alternate return from interrupt handler which BACKS UP PC,
||| and calls the scheduler prior to returning.  This causes
||| the trapped SVC to be re-executed when the process is
||| eventually rescheduled...

I_Wait:     LD(UserMState+(4*30), r0)      | Grab XP from saved MState,
      SUBC(r0, 4, r0)                | back it up to point to
      ST(r0, UserMState+(4*30))      |    SVC instruction

      CALL(Scheduler)                | Switch current process,
      BR(I_Rtn)               | and return to (some) user.
```

```
||| Sub-handler for SVCs, called from I_IllOp on SVC opcode:

SVC_UUO:
      LD(XP, -4, r0)            | The faulting instruction.
      ANDC(r0,0x7,r0)           | Pick out low bits,
      SHLC(r0,2,r0)             | make a word index,
      LD(r0,SVCTbl,r0)    | and fetch the table entry.
      JMP(r0)

SVCTbl:     UUO(HaltH)          | SVC(0): User-mode HALT instruction
      UUO(WrMsgH)         | SVC(1): Write message
      UUO(WrChH)          | SVC(2): Write Character
      UUO(GetKeyH)              | SVC(3): Get Key
      UUO(HexPrtH)              | SVC(4): Hex Print
      UUO(WaitH)          | SVC(5): Wait(S) ,,, S in R3
      UUO(SignalH)              | SVC(6): Signal(S), S in R3
      UUO(YieldH)         | SVC(7): Yield()

||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
||| Keyboard handling
||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||

Key_State: LONG(0)                    | 1-char keyboard buffer.

GetKeyH:                        | return key code in r0, or block
      LD(Key_State, r0)
      BEQ(r0, I_Wait)                 | on 0, just wait a while

| key ready, return it and clear the key buffer
      LD(Key_State, r0)         | Fetch character to return
      ST(r0,UserMState)         | return it in R0.
      ST(r31, Key_State)              | Clear kbd buffer
      BR(I_Rtn)                 | and return to user.


|||   Interrupt side: read key, store it into buffer.
|||   NB: This is a LIGHTWEIGHT interrupt handler, which doesn't
|||     do a full state save.  It doesn't have to, since (1) it
|||     only uses R0, and (2) it always returns to the same process
|||     it interrupts.  By not saving all state, it manages
|||     to save a LOT of time:  20 STs on entry, 30 LDs on exit:

KBD = 0xFFFFFFF8  | old value0xFFFFFFF8  | ps2 fifo

KbdState: LONG(0) | 0=normal, nonzero=release code
KbdModifier: LONG(0)    | which modifier keys are pressed
MOD_SHIFT_LEFT = 0x0100
MOD_SHIFT_RIGHT = 0x0200
MOD_CAPS_LOCK = 0x0400
MOD_SHIFT = 0x0700
MOD_CTRL = 0x0800 | should have separater flags for separate CTL keys
MOD_ALT = 0x1000  | should have separater flags for separate ALT keys

| scan in <7:0>, ascii in <15:8>, shifted in <23:16>
KbdScanTable:
      LONG(0x00333376)    | ESC
      LONG(0x00818105)    | F1
      LONG(0x00828206)    | F2
      LONG(0x00838304)    | F3
      LONG(0x0084840C)    | F4
      LONG(0x00858503)    | F5
      LONG(0x0086860B)    | F6
      LONG(0x00878783)    | F7
      LONG(0x0088880A)    | F8
      LONG(0x00898901)    | F9
      LONG(0x008A8A09)    | F10
```

```
      LONG(0x008B8B78)    | F11
      LONG(0x008C8C07)    | F12
      LONG(0x007E600E)    | backquote tilde
      LONG(0x00213116)    | 1 !
      LONG(0x0040321E)    | 2 @
      LONG(0x00233326)    | 3 #
      LONG(0x00243425)    | 4 $
      LONG(0x0025352E)    | 5 %
      LONG(0x005E3636)    | 6 ^
      LONG(0x0026373D)    | 7 &
      LONG(0x002A383E)    | 8 *
      LONG(0x00283946)    | 9 (
      LONG(0x00293045)    | 0 )
      LONG(0x005F2D4E)    | - _
      LONG(0x002B3D55)    | = +
      LONG(0x00080866)    | backspace
      LONG(0x0009090D)    | tab
      LONG(0x00517115)    | q Q
      LONG(0x0057771D)    | w W
      LONG(0x00456524)    | e E
      LONG(0x0052722D)    | r R
      LONG(0x0054742C)    | t T
      LONG(0x00597935)    | y Y
      LONG(0x0055753C)    | u U
      LONG(0x00496943)    | i I
      LONG(0x004F6F44)    | o O
      LONG(0x0050704D)    | p P
      LONG(0x007B5B54)    | [ {
      LONG(0x007D5D5B)    | ] }
      LONG(0x007C5C5D)    | \ |
      LONG(0x0041611C)    | a A
      LONG(0x0053731B)    | s S
      LONG(0x00446423)    | d D
      LONG(0x0046662B)    | f F
      LONG(0x00476734)    | g G
      LONG(0x00486833)    | h H
      LONG(0x004A6A3B)    | j J
      LONG(0x004B6B42)    | k K
      LONG(0x004C6C4B)    | l L
      LONG(0x003A3B4C)    | ; :
      LONG(0x00222752)    | ' "
      LONG(0x000A0A5A)    | enter
      LONG(0x005A7A1A)    | z Z
      LONG(0x00587822)    | x X
      LONG(0x00436321)    | c C
      LONG(0x0056762A)    | v V
      LONG(0x00426232)    | b B
      LONG(0x004E6E31)    | n N
      LONG(0x004D6D3A)    | m M
      LONG(0x003C2C41)    | , <
      LONG(0x003E2E49)    | . >
      LONG(0x003F2F4A)    | / ?
      LONG(0x00202029)    | space
      LONG(0x00909075)    | up arrow
      LONG(0x00919174)    | right arrow
      LONG(0x0092926B)    | left arrow
      LONG(0x00939372)    | down arrow

      LONG(0x00000000)    | scan code of 0 marks end of table

I_Kbd:      ENTER_INTERRUPT()        | Adjust the PC!
      ST(r0, UserMState)             | Save some regs
      ST(r1, UserMState+4)
      ST(r2, UserMState+8)
Kbd_loop:
      LD(KBD,r0)                | read scan code from fifo
      ANDC(r0,0x100,r1)         | check "empty" bit
```

```
      BEQ(r1,Kbd_process)              | if clear, process scan code
      LD(UserMState, r0)               | restore r0, and
      LD(UserMState+4, r1)             | restore r1, and
      LD(UserMState+8, r2)             | restore r2, and
      JMP(xp)                          | and return to the user.
Kbd_process:
      ANDC(r0,0xFF,r0)           | only keep 8-bit scan code
      CMPEQC(r0,0xE0,r1)               | throw away extend codes (for now)
      BT(r1,Kbd_loop)
      CMPEQC(r0,0xF0,r1)               | release code?
      BF(r1,Kbd_scan)
      ST(r0,KbdState)
      BR(Kbd_loop)
Kbd_scan:
      LD(KbdState,r2)                  | load up current state
      ST(r31,KbdState)           | reset state
      CMPEQC(r0,0x12,r1)               | shift? (left side)
      BT(r1,Kbd_shiftl)
      CMPEQC(r0,0x59,r1)               | shift? (right side)
      BT(r1,Kbd_shiftr)
      CMPEQC(r0,0x14,r1)               | control?
      BT(r1,Kbd_ctl)
      CMPEQC(r0,0x11,r1)               | alt?
      BT(r1,Kbd_alt)
      CMPEQC(r0,0x58,r1)               | caps lock?
      BT(r1,Kbd_capslock)
      BNE(r2,Kbd_loop)           | normal key => ignore release scan
      CMOVE(KbdScanTable,r1)           | look through scan table
Kbd_search:
      LD(r1,0,r2)                | load next entry from table
      ANDC(r2,0xFF,r2)           | mask off everything but scan code
      BEQ(r2,Kbd_loop)           | null table entry => end of table
      CMPEQ(r0,r2,r2)                  | match with incoming code
      BT(r2,Kbd_found)           | branch if a match
      ADDC(r1,4,r1)                    | increment table pointer
      BR(Kbd_search)                   | look in next entry
Kbd_found:
      LD(r1,0,r1)                | reload table entry
      SHRC(r1,8,r1)                    | ascii in <7:0>, shift in <15:8>
      LD(KbdModifier,r2)               | any shift keys pressed?
      ANDC(r2,MOD_SHIFT,r0)
      BEQ(r0,.+8)
      SHRC(r1,8,r1)                    | shift again if shift keys pressed
      ANDC(r1,0xFF,r1)           | just 8-bit result
      OR(r1,r2,r1)                     | add in modifier bits
      ST(r1,Key_State)           | save as most recent character
      BR(Kbd_loop)
Kbd_capslock:
      BEQ(r2,Kbd_loop)           | ignore key down for caps lock
      LD(KbdModifier,r2)               | caps lock toggles after each press
      ANDC(r2,MOD_CAPS_LOCK,r2)
      CMOVE(MOD_CAPS_LOCK,r0)
      BR(Kbd_modifier)
Kbd_shiftl:
      CMOVE(MOD_SHIFT_LEFT,r0)
      BR(Kbd_modifier)
Kbd_shiftr:
      CMOVE(MOD_SHIFT_RIGHT,r0)
      BR(Kbd_modifier)
Kbd_ctl:
      CMOVE(MOD_CTRL,r0)
      BR(Kbd_modifier)
Kbd_alt:
      CMOVE(MOD_ALT,r0)
Kbd_modifier:
      LD(KbdModifier,r1)               | current modifier state
      BNE(r2,Kbd_mod_clear)            | key up?
```

```
      OR(r1,r0,r1)
Kbd_mod_done:
      ST(r1,KbdModifier)
      BR(Kbd_loop)
Kbd_mod_clear:
      XORC(r0,-1,r0)
      AND(r1,r0,r1)
      BR(Kbd_mod_done)

WrChH:      LD(UserMState,r0)        | The user's <R0>
      CALL(KWrchar)                  | Write out the character,
      BR(I_Rtn)                | then return

WrMsgH:     LD(UserMState+(4*30), r0)      | Fetch interrupted XP, then
      CALL(KMsgAux)                  | print text following SVC.
      ST(r0,UserMState+(4*30))       | Store updated XP.
      BR(I_Rtn)

||| Handler for HexPrt(): print hex value from R0
HexPrtH:
      LD(UserMState,r0)        | Load user R0
      CALL(KHexPrt)                  | Print it out
      BR(I_Rtn)                | And return to user.

|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
||| Timesharing: 3-process round-robin scheduler
|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||

||| ProcTbl contains a 31-word data structure for each process,
|||  including R0-R30.  R31, which always contains 0, is omitted.
|||  The XP (R30) value stored for each process is the PC,
|||  and points to the next instruction to be executed.

||| The kernel variable CurProc always points to the ProcTbl entry
|||  corresponding to the "swapped in" process.

ProcTbl:
P0:   STORAGE(29)        | Process 0: R0-R28
P0SP: LONG(P0Stack)            | Process 0: SP
P0XP: LONG(P0Start)            | Process 0: XP (= PC)

P1:   STORAGE(29)        | Process 1: R0-R28
P1SP: LONG(P1Stack)            | Process 1: SP
P1XP: LONG(P1Start)            | Process 1: XP (= PC)

P2:   STORAGE(29)        | Process 2: R0-R28
P2SP: LONG(P2Stack)            | Process 2: SP
P2XP: LONG(P2Start)            | Process 2: XP (= PC)

CurProc: LONG(ProcTbl)

||| Schedule a new process.
||| Swaps current process out of UserMState, swaps in a new one.

Scheduler:
      PUSH(LP)
      CMOVE(UserMState, r0)
      LD(CurProc, r1)
      CALL(CopyMState)         | Copy UserMState -> CurProc

      LD(CurProc, r0)
      ADDC(r0, 4*31, r0)             | Increment to next process..
      CMPLTC(r0,CurProc, r1)         | End of ProcTbl?
      BT(r1, Sched1)                 | Nope, its OK.
      CMOVE(ProcTbl, r0)             | yup, back to Process 0.
Sched1:     ST(r0, CurProc)                | Here's the new process;
```

```
      ADDC(r31, UserMState, r1)     | Swap new process in.
      CALL(CopyMState)
      LD(Tics, r0)                  | Reset TicsLeft counter
      ST(r0, TicsLeft)        |   to Tics.
      POP(LP)
      JMP(LP)                       | and return to caller.

| Copy a 31-word MState structure from the address in <r0> to that in <r1>
| Trashes r2, leaves r0-r1 unchanged.
.macro CM(N) LD(r0, N*4, r2)  ST(r2, N*4, r1)   | Auxiliary macro
CopyMState:
      CM(0) CM(1) CM(2) CM(3) CM(4) CM(5) CM(6) CM(7)
      CM(8) CM(9) CM(10)      CM(11)      CM(12)      CM(13)      CM(14)      CM(15)
      CM(16)      CM(17)      CM(18)      CM(19)      CM(20)      CM(21)      CM(22)
      CM(23)
      CM(24)      CM(25)      CM(26)      CM(27)      CM(28)      CM(29)      CM(30)
      JMP(LP)

||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
||| Clock interrupt handler:  Invoke the scheduler.
||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||

||| Here's the deal:
||| Each compute-bound process gets a quantum consisting of TICS clock
|||   interrupts, where TICS is the number stored in the variable Tics
|||   below.  To avoid overhead, we do a full state save only when the
|||   clock interrupt will cause a process swap, using the TicsLeft
|||   variable as a counter.
||| We do a LIMITED state save (r0 only) in order to free up a register,
|||   then count down TicsLeft stored below.  When it becomes negative,
|||   we do a FULL state save and call the scheduler; otherwise we just
|||   return, having burned only a few clock cycles on the interrupt.
||| RECALL that the call to Scheduler sets TicsLeft to Tics, giving
|||   the newly-swapped-in process a full quantum.

CLKREG = 0xFFFFFFFC           | old value: 0xFFFFFFFC

Tics: LONG(2)                 | Number of clock interrupts/quantum.
TicsLeft: LONG(0)       | Number of tics left in this quantum

I_Clk:      ENTER_INTERRUPT() | Adjust the PC!
      LD(r31,CLKREG,r31)      | reset interrupt by reading CLK location
      ST(r0, UserMState)      | Save R0 ONLY, for now.
      LD(TicsLeft, r0)  | Count down TicsLeft
      SUBC(r0,1,r0)
      ST(r0, TicsLeft)  | Now there's one left.
      CMPLTC(r0, 0, r0) | If new value is negative, then
      BT(r0, DoSwap)          |   swap processes.
      LD(UserMState, r0)      | Else restore r0, and
      JMP(XP)                 | return to same user.

DoSwap:     LD(UserMState, r0)      | Restore r0, so we can do a
      SAVESTATE()       |   FULL State save.
      LD(KStack, SP)          | Install kernel stack pointer.
      CALL(Scheduler)         | Swap it out!
      BR(I_Rtn)         | and return to next process.


|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
||| yield() SVC: voluntarily give up rest of time quantum.
|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||

YieldH: CALL(Scheduler)       | Schedule next process, and
      BR(I_Rtn)         | and return to user.


||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
```

```
||| Here on start-up (reset):  Begin executing process 0.
|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||

I_Reset:
      LD(KStack, SP)           | Install kernel stack pointer.
      ST(r31,Key_State)
      ST(r31,KbdState)   | clear out keyboard state
      ST(r31,KbdModifier)

      CMOVE(P0Stack,r0) | initialize process table
      ST(r0,P0SP)
      CMOVE(P0Start,r0)
      ST(r0,P0XP)
      CMOVE(P1Stack,r0)
      ST(r0,P1SP)
      CMOVE(P1Start,r0)
      ST(r0,P1XP)
      CMOVE(P2Stack,r0)
      ST(r0,P2SP)
      CMOVE(P2Start,r0)
      ST(r0,P2XP)

      CMOVE(DPY,r0)            | clear display memory
clrloop:
      ST(r31,0,r0)
      ADDC(r0,4,r0)
      CMPLTC(r0,DPYEND,r1)
      BT(r1,clrloop)

      CMOVE(DPYLINE,r0) | initialize pointer into display buffer
      ST(r0,Kdpyptr)

      LD(Tics,r0)
      ST(r0,TicsLeft)
      CMOVE(P2,r0)             | start executing Process 2
      ST(r0,CurProc)
      CMOVE(P2Stack, SP)
      CMOVE(P2Start, XP)
      JMP(XP)

|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
||| SVC Sub-handler for user-mode HALTs
|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||

HaltH:      BR(I_Wait)              | SVC(0): User-mode HALT SVC


|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
||| Kernel support for User-mode Semaphores
|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||


||| User-mode access: macrodefinitions.    Semaphore adr passed in r3,
|||  which is saved & restored appropriately by macros:
||| NB: Wait() and Signal() SVCs each pass the address of a semaphore
|||  in R3.  Since the Illegal Opcode handler code doesn't change any
|||  registers except R0, the R3 semaphore address is still intact
|||  when we enter these handlers:

||| Kernel handler: wait(s):
||| ADDRESS of semaphore s in r3.

WaitH:      LD(r3,0,r0)        | Fetch semaphore value.
      BEQ(r0,I_Wait)           | If zero, block..

      SUBC(r0,1,r0)            | else, decrement and return.
      ST(r0,0,r3)        | Store back into semaphore
```

```
      BR(I_Rtn)          | and return to user.

||| Kernel handler: signal(s):
||| ADDRESS of semaphore s in r3.

SignalH:LD(r3,0,r0)            | Fetch semaphore value.
      ADDC(r0,1,r0)            | increment it,
      ST(r0,0,r3)        | Store new semaphore value.
      BR(I_Rtn)          | and return to user.

||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
||| Kernel-callable Utility Routines
||| NB: These routines use PRIVILEDGED instructions; hence they can be
|||  called directly only from kernel code (ie, with the high-PC-bit
|||  set).  Use SVC traps to accomplish the same functions from user-
|||  level code.
||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||

| simple display buffer management: always write characters into last line,
| scrolling up (by copying buffer) when we get a newline or write past the
| end of line.  Buffer stores 4 chars per word, so there are 20 words for
| each 80 character line.  There are 40 lines.

DPY = 0xFFFF8000

DPYEND = DPY + (80*40)
DPYLINE = DPY + (80*39)
Kdpyptr: LONG(DPYLINE)         | where we are in display buffer

KWrchar:
      PUSH(r0)
      PUSH(r1)
      PUSH(r2)
      CMPEQC(r0,'\n',r2)       | newline character?
      BT(r2,scroll)
      LD(Kdpyptr,r1)           | past end of line?
      CMPLTC(r1,DPYEND,r2)
      BT(r2,stchar)

scroll:
      CMOVE(DPY,r1)
sloop1:
      LD(r1,80,r2)             | grab word from next line
      ST(r2,0,r1)        | store in current line
      ADDC(r1,4,r1)            | bump pointer
      CMPLTC(r1,DPYLINE,r2)    | are we done?
      BT(r2,sloop1)            | nope, keep copying
sloop2:
      ST(r31,0,r1)             | clear last line
      ADDC(r1,4,r1)
      CMPLTC(r1,DPYEND,r2)
      BT(r2,sloop2)
      CMOVE(DPYLINE,r1)  | next char goes at beginning of last line

      CMPEQC(r0,'\n',r2)       | newline character?
      BT(r2,wrchar_rtn)

stchar:                        | char in R0, Kdpyptr in R1
      PUSH(r3)
      CMOVE(0xFF,r3)           | mask for faking STB
      ANDC(r1,3,r2)            | byte offset
      SHLC(r2,3,r2)            | multiply by 8 to get shift count
        SHL(r0,r2,r0)          | shift char/mask into correct position
      SHL(r3,r2,r3)
      LD(r1,0,r2)        | load word from display buffer
      XORC(r3,-1,r3)           | AND with complement of mask
      AND(r3,r2,r2)
```

```
      OR(r0,r2,r2)              | OR with shifted char
      ST(r2,0,r1)        | store back into display buffer
      ADDC(r1,1,r1)             | increment display pointer
      POP(r3)

wrchar_rtn:
      ST(r1,Kdpyptr,r31)        | save display pointer for next time
      POP(r2)
      POP(r1)
      POP(r0)
      RTN()              | return


||| Hex print procedure: prints longword in R0                  |||

HexDig:     LONG('0') LONG('1') LONG('2') LONG('3') LONG('4') LONG('5')
      LONG('6') LONG('7') LONG('8') LONG('9') LONG('A') LONG('B')
      LONG('C') LONG('D') LONG('E') LONG('F')

KHexPrt:
      PUSH(lp)
      PUSH(r0)           | Saves all regs, incl r0
      PUSH(r1)
      PUSH(r2)

      CMOVE(8, r2)
      MOVE(r0,r1)
KHexPr1:
      SRAC(r1,28,r0)                  | Extract digit into r0.
      SHLC(r1,4,r1)    |MULC(r1, 16, r1)| Next loop, next nybble...
      ANDC(r0, 0xF, r0)
      SHLC(r0,2,r0)    |MULC(r0, 4, r0)
      LD(r0, HexDig, r0)
      CALL(KWrchar)
      SUBC(r2,1,r2)
      BNE(r2,KHexPr1)

      POP(r2)
      POP(r1)
      POP(r0)
      POP(lp)
      RTN()

||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
||| Procedure to print out a zero-terminated message, packed one    |||
|||    char/byte. Char data follows branch; returns to next 4-byte   |||
|||    aligned location. Saves all regs.                    |||
||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||

KWrMsg:
      PUSH (R0)
      MOVE(LP, R0)
      CALL(KMsgAux)
      MOVE(R0, LP)
      POP (R0)
      RTN()


| Auxiliary routine for sending a message to the console.
| On entry, R0 should point to data; on return, R0 holds next
| longword aligned location after data.
| Note: Must be called while in supervisor mode.

KMsgAux:
      PUSH(lp)
      PUSH(r1)
      PUSH(r2)
```

```
      PUSH(r3)

      MOVE (R0, R1)

WrWord:     LD (R1, 0, R2)          | Fetch a 4-byte word into R2
      ADDC (R1, 4, R1)  | Increment word pointer
      CMOVE(4,r3)       | Byte/word counter

WrByte:     ANDC(r2, 0x7F, r0)      | Grab next byte -- LOW end first!
      BEQ(r0, WrEnd)          | Zero byte means end of text.
      CALL(KWrchar)           | Print it.
      SRAC(r2,8,r2)           | Shift out this byte
      SUBC(r3,1,r3)           | Count down... done with this word?
      BNE(r3,WrByte)          | Nope, continue.
      BR(WrWord)        | Yup, on to next.

WrEnd:
      MOVE (R1, R0)
      POP(r3)
      POP(r2)
      POP(r1)
      POP(lp)
      RTN()

||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
|||  User-mode code.  Includes 3 processes:
|||
|||  PROCESS 0:
|||    (1) Prompts the user for new lines of input.
|||    (2) Reads lines from the keyboard (using the GetKey() SVC),
|||        and pipes it to PROCESS 1 through a bounded buffer.
|||        It does this using the Send procedure.
|||
|||  PROCESS 1:
|||    Reads lines of input from PROCESS 0, using the Rcv procedure,
|||        translates them to Piglatin, and types them out (using
|||        the SVCs WrCh() and WrMsg().
|||
|||    Note that Send and Rcv, used by processes 0 and 1, communicate
|||        using a bounded buffer and synchronize using semaphores
|||        implemented as the Wait(S) and Signal(S) SVCs.
|||
|||  PROCESS 2:
|||    On each quantum, simply increments a counter and uses the Yield()
|||        SVC to give up the remainder of its quantum.  The resulting
|||        count thus becomes a count of the number of quanta which have
|||        been allocated to each process.  This count (in HEX) is used
|||        as the prompt typed by process 0. CCelio: also added Game of Life
|||
||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||

||| Definitions of macros used to interface with Kernel code:

.macro Halt()     SVC(0)            | Stop a process.

.macro WrMsg()    SVC(1)            | Write the 0-terminated msg following SVC
.macro WrCh()     SVC(2)            | Write a character whose code is in R0

.macro GetKey()   SVC(3)            | Read a key from the keyboard into R0
.macro HexPrt()   SVC(4)            | Hex Print the value in R0.

.macro Yield()    SVC(7)            | Give up remaining quantum

||| Semaphore macros.
||| Wait(S) waits on semaphore S; Signal(S) signals on S.
||| Both preserve all registers, by pushing & popping R3.
```

```
.macro Wait(S) {
      PUSH(r3)              | Save old <r3>,
      LDR(S,r3)             | put semaphore address into r3
      SVC(5)                      | Wait on semaphore whose adr is in R3
      POP(r3) }             | and restore former <r3>

.macro Signal(S) {
      PUSH(r3)              | Save old <r3>,
      LDR(S,r3)             | put semaphore address into r3
      SVC(6)                      | Signal on semaphore whose adr is in R3
      POP(r3) }             | and restore former <r3>

||| Allocate a semaphore: used like
|||    name:   semaphore(size)
.macro semaphore(N) {             | Allocate a semaphore, and build a ptr
   LONG(.+4)                      | Pointer to semaphore
   LONG(N) }                      | Semaphore itself, init value N.

|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
||| User-mode code: Process 0
|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||

Prompt:     semaphore(1)                | To keep us from typing next prompt
                            | while P1 is typing previous output.

P0Start:
      CMOVE(1,r0)
      ST(r0,Prompt+4)             | Initialize semaphores
      ST(r31,Chars+4)
      CMOVE(FIFOSIZE,r0)
      ST(r0,Holes+4)

      ST(r31,IN)            | Initialize circular buffer
      ST(r31,OUT)

      WrMsg()
      .text "Beta_OS v2.0.3: MultiCore GoL!\n\n"


P0Read: Wait(Prompt)              | Wait until P1 has caught up...
      WrMsg()                     | First a newline character, then
      .text "\n"
      LD(Count3, r0)              | print out the quantum count
      HexPrt()              |  as part of the count, then
       WrMsg()              |  the remainder.
      .text "> "

      LD(P0LinP, r3)              | ...then read a line into buffer...

P0RdCh: GetKey()            | read next character,
      ANDC(r0,0xFF,r0)
      WrCh()                      | echo back to user
      CALL(UCase)           | Convert it to upper case,
      ST(r0,0,r3)           | Store it in buffer.
      ADDC(r3,4,r3)               | Incr pointer to next char...

      CMPEQC(r0,0xA,r1) | End of line?
      BF(r1,P0RdCh)               | nope, keep filling buffer.

      LD(P0LinP,r2)               | Prepare to empty buffer.
P0PutC: LD(r2,0,r0)               | read next char from buf,
      CALL(Send)            | send to P2
      CMPEQC(r0,0xA,r1) | Is it end of line?
      BT(r1,P0Read)               | Yup, read another line.

      ADDC(r2,4,r2)               | Else move to next char.
      BR(P0PutC)
```

```
P0Line: STORAGE(100)           | Line buffer.
P0LinP: LONG(P0Line)

P0Stack:
      STORAGE(128)


||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
|||  Some auxilliaries for our little application:
||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||

| Auxilliary routine: convert char in r0 to upper case:
UCase:       PUSH(r1)
      CMPLEC(r0,'z',r1) | Is it beyond 'z'?
      BF(r1,UCase1)           | yup, don't convert.
      CMPLTC(r0,'a',r1) | Is it before 'a'?
      BT(r1, UCase1)          | yup, no change.
      SUBC(r0,'a'-'A',r0)     | Map to UPPER CASE...
UCase1: POP(r1)
      RTN()

| Auxilliary routine: Test if <r0> is a vowel; boolean into r1.
VowelP: CMPEQC(r0,'A',r1)     | Sorta brute force...
      BT(r1,Vowel1)
      CMPEQC(r0,'E',r1) BT(r1,Vowel1)
      CMPEQC(r0,'I',r1) BT(r1,Vowel1)
      CMPEQC(r0,'O',r1) BT(r1,Vowel1)
      CMPEQC(r0,'U',r1) BT(r1,Vowel1)
      CMPEQC(r0,'Y',r1) BT(r1,Vowel1)
      CMOVE(0,r1)       | Return FALSE.
Vowel1: RTN()

||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
|||  Bounded-buffer FIFO routines for Beta USER MODE
|||   CALL(Send) - sends datum in r0 thru pipe
|||   CALL(Rcv)    - reads datum from pipe into r0
||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||

FIFOSIZE = 100
FIFO: STORAGE(FIFOSIZE) | FIFO buffer.

IN:   LONG(0)                 | IN pointer: index into FIFO
OUT:  LONG(0)                 | OUT pointer: index into FIFO

Chars:       semaphore(0)             | Flow-control semaphore 1
Holes:       semaphore(FIFOSIZE)      | Flow-control semaphore 2

||| Send: put <r0> into fifo.
Send: PUSH(r1)          | Save some regs...
      PUSH(r2)
      Wait(Holes)       | Wait for space in buffer...

      LD(IN,r1)         | IN pointer...
      SHLC(r1,2,r2)           |MULC(r1,4,r2)    | Compute 4*IN, word offset
      ST(r0,FIFO,r2)          | FIFO[IN] = ch
      ADDC(r1,1,r1)           | Next time, next slot.
      CMPEQC(r1,FIFOSIZE,r2)  | End of buffer?
      BF(r2,Send1)            | nope.
      CMOVE(0,r1)       | yup, wrap around.
Send1:       ST(r1,IN)        | Tuck away input pointer

      Signal(Chars)           | Now another Rcv() can happen
      POP(R2)
      POP(r1)
      RTN()
```

```
||| Rcv: Get char from fifo into r0.

Rcv:  PUSH(r1)
      PUSH(r2)
      Wait(Chars)        | Wait until FIFO non-empty

      LD(OUT,r1)         | OUT pointer...
      SHLC(r1,2,r2)            | Compute 4*OUT, word offset
      LD(r2,FIFO,r0)           | result = FIFO[OUT]
      ADDC(r1,1,r1)            | Next time, next slot.
      CMPEQC(r1,FIFOSIZE,r2)   | End of buffer?
      BF(r2,Rcv1)        | nope.
      CMOVE(0,r1)        | yup, wrap around.
Rcv1: ST(r1,OUT)         | Tuck away input pointer

      Signal(Holes)            | Now theres space for 1 more.
      POP(R2)
      POP(r1)
      RTN()

||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
||| USER MODE Process 1: Translate English to Piglatin
||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||

|stay away from R9,23?
temp0 = R0
temp1 = R1
temp2 = R2
temp3 = R3
temp4 = R4
temp5 = R5
temp6 = R6
temp7 = R7
dy = R7
y_min = R8  |for the GoL, all cores loop thru x=[0,WIDTH), and y=[y_min, y_max)
y_max = R9
new_state = 10
core_count = R22
tempX = R11
tempY = R12

cpu_id = R13
i = R14
j = R15

new_c = R16
c = R17
n = R18
x = R19
y = R20
offset = R21

cellsA = R24
cellsB = R25
state = R26

SYNC_OFFSET = 32

||REVISION: only Core#0 echo's prompt to the screen
|| the other cores print out their ID and CORE_COUNT to denote their presence
|| BUG: it works the first time thru, but there is a bug in which after the first
|| time, only cores0 and 1 respond afterwards.  However, it does not freeze
|| the command_line (so it a "soft" bug)


P1Start:LD(P1BufP, r9)         | Buffer pointer in r9.
      CPUID(cpu_id)
```

```
      NCORES(core_count)       |Load number of cores into R12

P1Word: BEQ(cpu_id, P1Chars)   | begin immediately if CPU_ID# is zero

P1Wait: SUBC(cpu_id, 1, R6)    | Wait until previous CPU is ready....
      ADDC(R6,SYNC_OFFSET,R6)

WaitLoop:Yield()         | don't clog memory asking "are we there yet?"
      LD(R6, CPU_List, R1)     | See if Previous CPU is finished
      BF(R1, WaitLoop)
      NCORES(core_count)
      CMPLT(cpu_id,core_count, temp0)
      BF(temp0, WaitLoop)
      BR(P1EoL)

P1Chars:CALL(Rcv)        | Get next Character, or Wait for "ENTER" to be pressed
      CMPEQC(r0,0xA,r1)  | Was it end-of-line?
      BT(r1,P1EoL)
      WrCh()                   | Write the Character, and the loop for the next char
      BR(P1Chars)

P1EoL:      WrMsg()                  | append the EndOfLife fancy-stuff, like cpu_id,
ncore, etc.
      .text ": CPU #ID "
      MULC(cpu_id, 4, R0)      | this method only works for 16 cores
      LD(R0, HexDig, R0)
      WrCh()                   | Print ID#
      WrMsg()
      .text " of "
      NCORES(core_count)       |refresh core_count
      MULC(core_count,4,R0)
      LD(R0, HexDig, R0)       | this only works for <16 cores
      WrCh()                   | Print N_CORES
      |MOVE(r3,r0)             | Print out the EoL Character ("\n")
      CMOVE(0xA, r0)
      WrCh()

P1CheckIn:
      CMOVE(1, R0)             |Load "TRUE" into R0
      ADDC(cpu_id, SYNC_OFFSET, R1) | calculate index for array
      ST(R0, CPU_List, R1)     |"Check In"
      NCORES(core_count)
      SUBC(core_count, 1, R1)
      CMPEQC(cpu_id, R1, R0)   | Are you the Last CPU?
      BF(R0, P1WaitEnd)

P1LastCPU:               | 1. Clear CPU_CheckIn list
                         | 2. Signal prompt to delete line data.
      CMOVE(0, i)        | i = 0
P1forBegin:
      ADDC(i, SYNC_OFFSET, R1)
      ST(R31, CPU_List, R1)    | Clear elementyes,

      ADDC(i, 1, i)            | i++
      |CMPLT(i, core_count, R1)| i < n_cores
      CMPLTC(i,16,R1)          |***MAGIC NUMBER*** core_count will actually be buggy
here
                         |problem arises if you have more cores than core_count running
                         |and you don't clear their check-in slots
      BT(R1, P1forBegin)

P1forEnd:
      Signal(Prompt)           | allow proc 0 to re-prompt.
      BR(P1Word)         | ... and start another word.


P1WaitEnd:               | CPU has finished printing, wait for their spot to be cleared
```

```
      Yield()                  |Yield Time to stop clogging memory
      ADDC(cpu_id, SYNC_OFFSET,R1)
      LD(R1, CPU_List, R0)
      BT(R0, P1WaitEnd) | while(CPU_List[cpu_id]==1) { }
      Signal(Prompt)           | allow proc 0 to re-prompt.
      BR(P1Word)         | ... and start another word.

CPU_List = 0xFFFFFF04

P1Buf:      STORAGE(100)             | Line buffer.
P1BufP: LONG(P1Buf)            | Address of line buffer.
P1Stack: STORAGE(256)          | Stack for process 2.

||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
||| USER MODE Process 2: Simply counts quanta & Game of Life!
||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||


HEIGHT = 120      |this *MUST* match the display dimensions in the
WIDTH = 120 | display controller module in verilog

CellsAAddr= 0xFFFF8d00
CellsBAddr= 0xFFFFc600
StateAddr= 0xFFFFFF00


P2Start:
      ST(r31,Count3)
      CPUID(cpu_id)            |load the CPU's ID#
      CMOVE(0, state)          |initialize state_buffer to be zero

P2Loop:     |silly vestigal code....
      LD(Count3, r0)           | Another quantum, incr count3.
      ADDC(r0,1,r0)
      ST(r0,Count3)


GameOfLife:
      CMOVE(0, x)        | loop width, int i=0
      NCORES(core_count)       | update core_count every round, so
                         | user can change the number of CPU's
      LD(StateAddr, state)     |

      CMPLT(cpu_id, core_count, temp1)| coreID# < CORE_COUNT
      BF(temp1, GoLIdle)       |if you're not suppose to be running, go to IDLE

CalcDy: CMPEQC(core_count, 1, temp0)| if(core_count==1) -> etc.
      BT(temp0, _1_core)
      CMPEQC(core_count, 2, temp0)| if(core_count==2) -> etc.
      BT(temp0, _2_core)
      CMPEQC(core_count, 3, temp0)| if(core_count==3) -> etc.
      BT(temp0, _3_core)
      CMPEQC(core_count, 4, temp0)| if(core_count==4) -> etc.
      BT(temp0, _4_core)
      CMPEQC(core_count, 5, temp0)| if(core_count==5) -> etc.
      BT(temp0, _5_core)
      CMPEQC(core_count, 6, temp0)| if(core_count==6) -> etc.
      BT(temp0, _6_core)
      CMPEQC(core_count, 7, temp0)| if(core_count==7) -> etc.
      BT(temp0, _7_core)
      CMPEQC(core_count, 8, temp0)| if(core_count==8) -> etc.
      BT(temp0, _8_core)

_1_core: CMOVE(HEIGHT,dy)
      BR(CalculateYBounds)
_2_core: CMOVE(HEIGHT/2,dy)
      BR(CalculateYBounds)
```

```
_3_core: CMOVE(HEIGHT/3,dy)
      BR(CalculateYBounds)
_4_core: CMOVE(HEIGHT/4,dy)
      BR(CalculateYBounds)
_5_core: CMOVE(HEIGHT/5,dy)
      BR(CalculateYBounds)
_6_core: CMOVE(HEIGHT/6,dy)
      BR(CalculateYBounds)
_7_core: CMOVE(HEIGHT/7,dy)   |this is the only one that doesnt' divide 120 cleanly
==17
      BR(CalculateYBounds)
_8_core: CMOVE(HEIGHT/8,dy)

CalculateYBounds:
      MUL(cpu_id, dy, y_min)
      ADD(y_min, dy, y_max)

x_loop:
      |ADDC(y_min, 0, temp0)
      MOVE(y_min, y)          | loop height, int y = y_min

y_loop:                 | get current cell's value
                        | calculate offset, ie, cell[i,j]
GetCell:MULC(y, HEIGHT, offset)     | offset = j*HEIGHT + i
      ADD(offset, x, offset)

state0_if:
      BT(state, LoadB)  |if(state==0) -> LoadA else LoadB
LoadA:  LD(offset, CellsAAddr, c)|  c = CellsA[x,y]
      BR(CountNeighbors)
LoadB:      LD(offset, CellsBAddr, c)|    c = CellsB[x,y]

CountNeighbors:
      CMOVE(0,n)        | int n = 0 (neighbor_count)
beginfor_i:
      CMOVE(-1,i)
i_loop:
      CMOVE(-1,j)
j_loop:
      ADD(i,x,tempX)          |calculate offset for x, y, ie, tempX,tempY
      ADD(j,y,tempY)

validloc_if:
      CMPLT(R31,tempX,temp1)  | if(tempX > 0          (longest IF statementEVAR)
      CMPLTC(tempX,WIDTH,temp2)|    && tempX < WIDTH
      CMPLT(R31,tempY,temp3)  |     && tempY > 0
      CMPLTC(tempY,HEIGHT,temp4)|   && tempY < HEIGHT
                        |
      AND(temp1,temp2,temp0)
      AND(temp3,temp4,temp1)
      AND(temp1,temp0,temp0)

      CMPEQ(R31,i,temp1)      |     &&
      CMPEQ(R31,j,temp2)      |     !(i==0 && j==0)
      AND(temp1,temp2,temp1)
      XORC(temp1,1,temp1)     |     !(stuff)

      AND(temp0,temp1,temp0)  |     (FINISH ANDING EVERYTHING)
      BF(temp0,j_endfor)      | if x,y are not valid locations,
                        | then do NOT count "n"

      MULC(tempY, HEIGHT, temp0)| calculate offset= tempY*HEIGHT + tempX
      ADD(temp0, tempX, temp0)

state1_if:
      BT(state,countB)  | if(state==0) -> CountA else CountB
countA: LD(temp0,CellsAAddr,temp1)
```

```
      BR(state1_endif)
countB: LD(temp0,CellsBAddr,temp1)

state1_endif:
      ADD(temp1,n,n)           | n = n + neighborCell


j_endfor:
      ADDC(j,1,j)
      CMPLEC(j,1,temp0)
      BT(temp0, j_loop)

i_endfor:
      ADDC(i,1,i)
      CMPLEC(i,1,temp0)
      BT(temp0, i_loop)


CalculateC:
      BT(c, LiveCell_if)       |if (cell==0) deadCell, ELSE-> LiveCell

DeadCell:_if:
      CMPEQC(n,3,temp0)
      BF(temp0, DeadCell_else)
      CMOVE(1,new_c)           |make cell alive!
      BR(endCalculateC)

DeadCell_else:
      CMOVE(0,new_c)           |keep cell dead
      BR(endCalculateC)

LiveCell_if:                   | if(n!=2 && n!=3) cell is killed (make it dead)
      CMPEQC(n,2,temp1)
      CMPEQC(n,3,temp2)
      OR(temp1,temp2, temp0)   | !(n==2 || n==3)
      XORC(temp0,1,temp0)      | invert


      BF(temp0, LiveCell_else) |if !(n==2 || n==3) -> cell_killed else ->  cell_alive
      CMOVE(0,new_c)
      BR(endCalculateC)

LiveCell_else:
      CMOVE(1,new_c)


endCalculateC:

|state2_if:
      BF(state,storeB)  | if(!state) -> StoreB else StoreA
storeA: ST(new_c,CellsAAddr,offset)
      BR(state2_endif)
storeB: ST(new_c,CellsBAddr,offset)
state2_endif:


|y_endfor:
      ADDC(y, 1, y)            | y++
      SUBC(y_max, 0, temp0)    | this is just debug offset, to not go to borders
      CMPLT(y, temp0, R0)
      BT(R0, y_loop)           | loop again

x_endfor:
      ADDC(x, 1, x)      | i++
      CMPLTC(x, WIDTH, R0)     | x < WIDTH
      BT(R0, x_loop)           | loop again
```

```
|**********************************
 Finished with calculating new field
| now, time to check-in and synchronize all the betas

      XORC(state, 1, new_state) | new_state = ~state;

GoLCheckIn:
      CMOVE(1, R0)              |Load "TRUE" into R0
      ST(R0, CPU_List, cpu_id)|"Check In"
      CMPEQC(cpu_id, 0, R0)     | Are you the CPU#0?
      BF(R0, GoLWaitEnd)


CPU0WaitforCheckIn:             |CPU#0 waits for all other cores to check in
      CMOVE(1,i)
CPU0WhileLoop:                  | while(i < n_cores)
      LD(i, CPU_List,temp0)|
      BF(temp0,CPU0WhileEnd)    | if(CPU_List[i+core_count]==1)
      ADDC(i,1,i)         |     i++;
CPU0WhileEnd:                   |
      CMPLT(i,core_count,temp0)
      BT(temp0,CPU0WhileLoop)

                          | 1. Clear CPU_CheckIn list
                          | 2. Change "state" to signal Display Controller
                          |     and cores to begin new round

      CMOVE(0, i)         | i = 0
GoLClearSyncforBegin:
      |ADD(i, core_count, temp1) |calculate offset to access CPU_list
      ST(R31, CPU_List, i) | Clear elementyes,

      ADDC(i, 1, i)             | i++
      CMPLT(i, core_count, R1)| i < n_cores *** changed from C(i,NCORES)....
      BT(R1, GoLClearSyncforBegin)

      MOVE(new_state, state)
      ST(new_state, StateAddr)| cpu#0 should write to this for two reasons:
                          | 1. signal Display Controller to read the new GoLbuffer
                          | 2. signal all cores to start a new round

GoLWaitEnd:               | all cores!=0 need to wait when finished
                          | they wait for CPU#0 to change the "state" register
                          | this doesn't go thru the arbiter, so this
                          | doesn't clog shared memory =)
      LD(StateAddr, state)
      CMPEQ(state, new_state, temp0)
      BF(temp0, GoLWaitEnd)


|**********************************
                          | use this to throttle the speed of the game
      BR(P2Loop)          | return here after others run.

GoLIdle:                  | if your cpu_id >= core_count, you are
                          | not allowed to play (for real-time changing core_count
                          | therefore, they come here instead.
      Yield()
      BR(P2Loop)

P2Stack: STORAGE(128)

Count3: LONG(0)

||||
|| TO DO:
|| get command-line synchronized
```

BUGS:
hitting "enter" freezes synchro in P1, in short, i suspect
other processes aren't getting character input

Game_Of_Life.c

```
/* Written by Christopher Celio, April 2007
      This code may be inaccurate or flawed, and
      it certainly is not complete, but it was used
      as a template for writing the GoL in assembly,
      and was highly successful in that regard.

      This document should be used as a reference in
      understanding the logic flow for the GoL, and
      to help understand the the assembly version
      found in Beta OS 2.0.3
*/

int GameOfLife() {

      int width = 120;
      int height = 120;

      int cells1 [width][height]; //matrices in the GoL Memory module
      int cells2 [width][height];

                              //state is the Buffer_Select_Register
      int state = 0;    //it toggle between reading cells1, and writing to cells2
                              // and reading cells2, and writing to cells1
                              // this prevents cpu1 from writing data before cpu2 has
used it

      int cpu_id = GET_CPU_ID;            //new primitive instructions
      int n_cores = GET_CORE_COUNT;

      int CPU_checkin[n_cores]; //This is stored in the Sync RAM module



      //Let's play the Game of Life!
      while(true) {

      //I can't divide, so (height/n_cores) is calculatd by using a case_select
      //statement in assembly.
      int y_min = cpu_id * (height / n_cores);
      int y_max = y_min + (height / n_cores);

      //loop thru the entire field
      for(i=0; i < width; i++) {

            for(j= y_offset; j < y_offset + (height/n_cores); j++) {

                  //1. get current cell situation
                  if(state==0)
                        c = cells1[i][j];
                  else
                        c = cells2[i][j];


                  //2. calculate number of neighbors
                  //getNumberOfNeighbors(i,j);
                  n = 0;

                  //can of course be optimized by not doing this with a for loop!
                  for(x = -1; x <=1; x++) {
                        for(y = -1; y <=1; y++) {
                              if(x > 0
```

```
                              && x < width
                              && y > 0
                              && y < height
                              && !(i==x && j==y))

                              if(state==0)
                                    n = n + cells1[i][j];
                              else
                                    n = n + cells2[i][j];

                  }
            }


            //3. figure out if cell should live, die, or be born
            if(c==0) { //cell is dead
                  if(n==3)
                        //cell is born
                        if(state==0)
                              cells2[i][j] = 1;
                        else
                              cells1[i][j] = 1;
            }
            else {//cell is alive
                  if(n!=2 && n!=3)
                        //cell is killed
                        if(state==0)
                              cells2[i][j]=0;
                        else
                              cells1[i][j]=0;

            }


      }

}


/* 4. Finished looping through their domain.
      Now, synchronize CPU's, switch states, and continue again!
*/

//A: check-in, this is a write to the Sync RAM
new_state = ~state;
CPU_checkin[cpu_id] = 1;

/* B. Let CPU #0 wait for all CPU's to check in,
      clear list, change state.
*/
if(cpu_id = 0) {

      //check if everyone is finished
      i =0;
      while(i < n_cores) {
            if(CPU_checkin[i] == 1)
                  i++;
      }

      //clear list
      for(i=0; i < n_cores; i++) {
            CPU_checkin[i] = 0;
      }

      state = new_state;
      //this signals to all the other processors to start the next round!
}
```

```
    else {
          //not cpu #0, wait until the state has changed.

          while(state!=new_state) {
                /*    wait here.
                      normally, this would clog
                      shared_memory resources, so
                      we pipe state (i.e.,Buffer Select Register)
                      directly to all Betas
                */
          }
    }

    //Begin next Round!
    }

}
```

BetaMem_Script.py (Adapted from C. Terman's script)

```
#!/usr/bin/env python
import sys,os,os.path,traceback

# get name of code/module
if (len(sys.argv) != 1):   #Matt Long : changed from 2 -> 1
    print "Usage: betamem <modulename>"
    sys.exit(0)

mname = "Beta_OS_203f" # sys.argv[1]  Matt Long : specified name of coe file, this was
the last change made
numfiles = 16 #Matt: specify how many files to create, each will have different module
names

# read in memory contents
coename = mname + ".coe"
if not os.path.exists(coename):
    print "Oops: can't find %s" % coename
    sys.exit(0)

try:
    f = open(coename)
    contents = f.read()  # read in entire file
    f.close()

except Exception,e:
    print "Oops:",e
    sys.exit(0)

# make a list, one entry per location.  Skip past
# any coe header lines.

contents = contents.replace(',','').split('\n')

# convert each hex string to an integer
locations = []
for line in contents:
    if len(line) == 0: continue
    elif line[0] == 'm': continue
    try:
        line = line.replace(';','')
        locations.append(int(line,16))
    except Exception,e:
```

```
        print "Oops: error reading location",(len(locations)+1),": ",e
        sys.exit(0)
nlocs = len(locations)

# helper function returns binary string with WIDTH
# digits from BITOFFSET within location LOCN
def bits(width,bitoffset,locn):
    if locn >= nlocs: v = 0
    else: v = locations[locn]
    v >>= bitoffset;
    result = []
    for i in xrange(width):
        if v % 2 == 0: result.append('0')
        else: result.append('1')
        v >>= 1
    result.reverse()
    return ''.join(result)

# see what BRAM organization to use
if (nlocs <= 512):
    nmems = 1            # use a single 512 x 36 BRAM
    bram = "RAMB16_S36"
    naddr = 9
    width = 32
    pwidth = 4

elif (nlocs <= 1024):
    nmems = 2            # use two 1024 x 16 BRAMs
    bram = "RAMB16_S18"
    naddr = 10
    width = 16
    pwidth = 2

elif (nlocs <= 2048):
    nmems = 4            # use four 2048 x 8 BRAMs
    bram = "RAMB16_S9"
    naddr = 11
    width = 8
    pwidth = 1

elif (nlocs <= 4096):
    nmems = 8            # use eight 4096 x 4 BRAMs
    bram = "RAMB16_S4"
    naddr = 12
    width = 4
    pwidth = 0

elif (nlocs <= 8192):
    nmems = 16           # use sixteen 8192 x 2 BRAMs
    bram = "RAMB16_S2"
    naddr = 13
    width = 2
    pwidth = 0

elif (nlocs <= 16384):
    nmems = 31           # use thirty-two 16384 x 1 BRAMs
    bram = "RAMB16_S1"
    naddr = 14
    width = 1
```

```
    pwidth = 0

else:
    print "Oops: %d is too big, can only support up to 16k locations" % nlocs
    sys.exit(0)

# ready to create appropriate Verilog module
unique_num = 0
try:
    for i in xrange(numfiles):
        current = i
        vname = mname + str(i) + "test.v"
        print "Making %s" % vname
        v = open(vname,'w')

        # output standard module prologue
        v.write("""// single-port read/write memory initialized with %s code

module %s(addr,clk,din,dout,we);
  input [13:0] addr;     // up to 16K locations
  input clk;             // memory has internal address regs
  input [31:0] din;      // appears after rising clock edge
  output [31:0] dout;    // written at rising clock edge
  input we;              // enables write port

  // we're using %d out of %d locations
""" % (mname + str(i),mname + str(unique_num)+"test",nlocs,1 << naddr))

        # output appropriate number of BRAM instances
        for i in xrange(nmems):
            lo = i * width
            hi = lo + width - 1
            if pwidth > 0:
                parity = ".DIP(%d'h0)," % pwidth
            else:
                parity = ""
            unique_num = unique_num + 1
            v.write("  %s
m%s(.CLK(clk),.ADDR(addr[%d:0]),.DI(din[%d:%d]),%s.DO(dout[%d:%d]),.WE(we),.EN(1'b1),.S
SR(1'b0));\n" % (bram,str(unique_num)+"test",naddr-1,hi,lo,parity,hi,lo))

            # output defparams to initialize this BRAM block
            nwords = 256/width
            for init in xrange(64):
                v.write("  defparam m%s.INIT_%02X = 256'b" %
(str(unique_num)+"test",init))
                start = init * nwords
                first = True
                for locn in xrange(start+nwords,start,-1):
                    if first: first = False
                    else: v.write('_')
                    v.write(bits(width,lo,locn-1))
                v.write(';\n')

        v.write("\nendmodule")
        v.close()

except Exception,e:
    print "Oops:",e
```

```
    sys.exit(0)

print "Done!"
# finished!
```

FONTMEM.py (Thanks to C. Terman)

```
#!/usr/bin/pyton
import sys,os,os.path,traceback

mname = 'font'

# read in memory contents
coename = mname + ".coe"
if not os.path.exists(coename):
    print "Oops: can't find %s" % coename
    sys.exit(0)
try:
    f = open(coename)
    contents = f.read()  # read in entire file
    f.close()
except Exception,e:
    print "Oops:",e
    sys.exit(0)

# make a list, one entry per location.  Skip past
# any coe header lines.
contents = contents.replace(',','').split('\n')

# convert each binary string to an integer
locations = []
for line in contents:
    if len(line) == 0: continue
    elif line[0] == 'm': continue
    try:
        line = line.replace(';','')
        locations.append(int(line,2))
    except Exception,e:
        print "Oops: error reading location",(len(locations)+1),": ",e
        sys.exit(0)
nlocs = len(locations)

# helper function returns binary string with WIDTH
# digits from BITOFFSET within location LOCN
def bits(width,bitoffset,locn):
    if locn >= nlocs: v = 0
    else: v = locations[locn]
    v >>= bitoffset;
    result = []
    for i in xrange(width):
        if v % 2 == 0: result.append('0')
        else: result.append('1')
        v >>= 1
    result.reverse()
    return ''.join(result)

# ready to create appropriate Verilog module
try:
    vname = 'xfont.v'
    v = open(vname,'w')
```

```
    # output standard module prologue
    v.write("""// 8x12 font memory for 128 chars
module xfont(addr,clk,row);
  input clk;
  input [10:0] addr;
  output [7:0] row;

  // font read-only memory: (128 * 12row/chars) x (8 bits/row)
  RAMB16_S9 font(.CLK(clk),.ADDR(addr),.DO(row),
                 .WE(1'b0),.EN(1'b1),.SSR(1'b0));
""")

    nwords = 256/8
    lo = 0;
    width = 8;
    for init in xrange(64):
        v.write("  defparam font.INIT_%02X = 256'b" % (init))
        start = init * nwords
        first = True
        for locn in xrange(start+nwords,start,-1):
            if first: first = False
            else: v.write('_')
            v.write(bits(width,lo,locn-1))
        v.write(';\n')

    v.write("\nendmodule")
    v.close()
except Exception,e:
    print "Oops:",e
    sys.exit(0)

# finished!
```