Contents

internal.v
internal....2
add_rgb....3
controller....3
delay....4
vga....4

screen_buffer.v
screen_buffer....5
zbt_out....6
zbt_reader....7

triangle_shader.v
triangle_shader....8
point_to_pixel....9
rhombus_shader....9
point_left_of_line....10

triangle_pipeline.v
triangle_pipeline....11
keep_visible_block0....12
keep_visible_block1....12
light_block0....12
light_block1....12
normal_block....13
opt_block....13
opt_translation_block....13
pipeline_register....13
reorder_block....13
rotation_block....14
translation_block....14
zoom_block....14

triangle_source.v
triangle_source....15
rom_data_source....15

orienter.v
orienter....16
slider....17

game_fsm.v
game_fsm....18
ball_fsm....19
paddle_fsm....19

display_field.v
display_field....20
box....20

signed.v
signed_add....21
signed_add3....21
signed_inc....21
signed_lt....21
signed_bound....21
signed_multiply....21
signed_negate....21
signed_outer3....21
signed_subtract....21
signed_to_unsigned....21
signed_trim....21

vector.v
vector_subtract....22
vector_matrix_product....22
vector_trim....22
dot_product....22
cross_product....22

matrix.v
matrix_product....23
matrix_subtract....23
matrix_trim....23
matrix_vector_product....23

thetas_to_matrix.v
thetas_to_matrix....24
theta_to_matrix....25
divider....25

labkit.v
labkit....25
debounce....27

internal_tf.v
internal_tf_v....28

defines.v....29

params.v....30

internal.v

```
`timescale 1ns / 1ps
`include "defines.v"
module internal(
    reset, clock,

    model_select,

    paddle_up, paddle_down, paddle_speed, ball_initial_speed, // game inputs

    ram0_data, ram0_address, ram0_we_b, // ram 0 buss
    ram1_data, ram1_address, ram1_we_b, // ram 1 buss


    mouse0_clock, mouse0_data, // ps/2 0 buss
    mouse1_clock, mouse1_data, // ps/2 1 buss
    buttons,

    vga_rgb, vga_blank_b, vga_hsync, vga_vsync // vga outputs
    );

    input clock;
    input reset;

    input [3:0] model_select;

    input paddle_up;
    input paddle_down;
    input [3:0] paddle_speed;
    input [3:0] ball_initial_speed;

    inout [35:0] ram0_data;
    output [18:0] ram0_address;
    output ram0_we_b;

    inout [35:0] ram1_data;
    output [18:0] ram1_address;
    output ram1_we_b;

    input mouse0_clock;
    input mouse0_data;
    input mouse1_clock;
    input mouse1_data;
    input [`NUM_BUTTONS-1:0] buttons;

    output [23:0] vga_rgb;
    output vga_blank_b;
    output vga_hsync;
    output vga_vsync;

    wire [23:0] read_rgb, df_rgb;
    wire blank_b, hsync, vsync, enable, blanking;
    wire [9:0]  pixel_count, line_count;
    vga vga1(
        .reset(reset),
        .pixel_clock(clock),
        .blank_b(blank_b),
        .hsync(hsync),
        .vsync(vsync),
        .pixel_count(pixel_count),
        .line_count(line_count),
        .blanking(blanking),
        .enable(enable));

```

Page: 1

internal.v

```
    delay #(5) dy1 (reset, clock, blank_b, vga_blank_b);
    delay #(5) dy2 (reset, clock, hsync, vga_hsync);
    delay #(5) dy3 (reset, clock, vsync, vga_vsync);

    wire [9:0] paddle_y;
    wire [9:0] ball_x;
    wire [9:0] ball_y;
    display_field df
        (reset, pixel_count, line_count,
         paddle_y, ball_x, ball_y, 24'b0, df_rgb);
    add_rgb #(0, 1) ar
        (reset, clock, read_rgb,
        df_rgb & (~{(24){model_select[0]||model_select[1]}}), vga_rgb);

    wire switch_buffer;
    wire [9:0] write_pixel_count, write_line_count;
    wire [11:0] write_z;
    wire [23:0] write_rgb;
    wire pixel_next;
    screen_buffer sb(
        .reset(reset),
        .clock(clock),
        .switch_buffer(switch_buffer),

        .write_pixel_count(write_pixel_count),
        .write_line_count(write_line_count),
        .write_z(write_z),
        .write_rgb(write_rgb),
        .write_noop(pixel_noop),
        .write_next(pixel_next),

        .read_pixel_count(pixel_count),
        .read_line_count(line_count),
        .read_rgb(read_rgb),

        .ram0_data(ram0_data),
        .ram0_address(ram0_address),
        .ram0_we_b(ram0_we_b), // ram 0 buss

        .ram1_data(ram1_data),
        .ram1_address(ram1_address),
        .ram1_we_b(ram1_we_b) // ram 1 buss
        );

    wire [`TRIANGLE_BITS+`NORMAL_BITS-1:0] triangle_data;
    wire triangle_empty, triangle_noop, triangle_next, next_frame;
    triangle_source ts (reset,
                        clock,
                        model_select,
                        triangle_data,
                        triangle_empty,
                        triangle_noop,
                        triangle_next,
                        next_frame);


    controller cr(
        .reset(reset),
        .clock(clock),
        .blanking(blanking),
        .empty(pixel_empty),
        .next_frame(next_frame),
        .switch_buffer(switch_buffer));
```

Page: 2

```



    fsm fsm1 (.reset(reset),
              .clock(clock),
              .enable(enable),
              .speed(ball_initial_speed),
              .paddle_speed(paddle_speed),
              .up_sync(paddle_up),
              .down_sync(paddle_down),
              .paddle_y(paddle_y),
              .ball_x(ball_x),
              .ball_y(ball_y));

    wire [`COORD_BITS*3-1:0] translation_vector, light_vector,
                             ball_vector, paddle_vector;
    wire [9*`TRIG_OUT_BITS-1:0] rotation_matrix, rotor_matrix;
    wire [`COORD_BITS-1:0] zoom;
    orienter ornt(
        reset,
        clock,
        mouse0_clock, mouse0_data, mouse1_clock, mouse1_data, buttons,
        enable, switch_buffer,
        paddle_y, ball_x, ball_y,
        translation_vector,
        rotation_matrix,
        light_vector,
        zoom,
        rotor_matrix,
        ball_vector,
        paddle_vector);

    wire [`TRIANGLE_BITS-1:0] out_triangle;
    wire out_noop, out_empty, out_next;

    triangle_pipeline tp(
        .reset(reset),
        .clock(clock),
        .in_triangle(triangle_data),
        .in_noop(triangle_noop),
        .in_empty(triangle_empty),
        .in_next(triangle_next),
        .out_triangle(out_triangle),
        .out_noop(out_noop),
        .out_empty(out_empty),
        .out_next(out_next),
        .translation_vector(translation_vector),
        .rotation_matrix(rotation_matrix),
        .light_vector(light_vector),
        .zoom(zoom),
        .rotor_matrix(rotor_matrix),
        .ball_vector(ball_vector),
        .paddle_vector(paddle_vector));

    triangle_shader pshdr(
        .reset(reset),
        .clock(clock),
        .triangle_data(out_triangle),
        .triangle_empty(out_empty),
        .triangle_noop(out_noop),
        .triangle_next(out_next),
        .pixel_data({write_pixel_count,
                     write_line_count,
```



```
                    write_z, write_rgb}),
        .pixel_empty(pixel_empty),
        .pixel_noop(pixel_noop),
        .pixel_next(pixel_next));
endmodule

module add_rgb(reset, clock, a, b, c);
    parameter A_SHIFT = 1;
    parameter B_SHIFT = 1;

    localparam T_WIDTH = 9;
    input reset, clock;
    input [23:0] a, b;
    output [23:0] c;

    reg [23:0] c;
    wire [23:0] next_c;
    wire [7:0] a_r, a_g, a_b,
               b_r, b_g, b_b,
               c_r, c_g, c_b;

    assign {a_r, a_g, a_b} = a;
    assign {b_r, b_g, b_b} = b;
    assign next_c = {c_r, c_g, c_b};

    wire [T_WIDTH-1:0] t_r, t_g, t_b;

    assign t_r = a_r[7-A_SHIFT:0] + b_r[7-B_SHIFT:0];
    assign t_g = a_g[7-A_SHIFT:0] + b_g[7-B_SHIFT:0];
    assign t_b = a_b[7-A_SHIFT:0] + b_b[7-B_SHIFT:0];

    assign c_r = t_r > 255 ? 255 : t_r[7:0];
    assign c_g = t_g > 255 ? 255 : t_g[7:0];
    assign c_b = t_b > 255 ? 255 : t_b[7:0];

    always @(posedge clock)
        if (reset)
            c <= 24'b0;
        else
            c <= next_c;
endmodule

module controller(reset, clock, blanking, empty, next_frame, switch_buffer);
    input reset;
    input clock;
    input blanking;
    input empty;
    output next_frame;
    output switch_buffer;

    reg [2:0] state;
    localparam IDLE =   3'd0;
    localparam WAIT_FOR_BLANKING = 3'd1;
    localparam SEND_SIGNALS = 3'd2;
    localparam NEITHER_LOW = 3'd3;
    localparam EMPTY_LOW = 3'd4;
    localparam BLANKING_LOW = 3'd5;

    reg [2:0] next_state;

    always @ (posedge clock)
        if (reset)
            state <= IDLE;
```

```
        else
            state <= next_state;

    always @ (state or blanking or empty) begin
        next_state = state;
        case (state)
            IDLE:
                if (empty)
                    if (blanking)
                        next_state = SEND_SIGNALS;
                    else
                        next_state = WAIT_FOR_BLANKING;


            WAIT_FOR_BLANKING:
                if (blanking)
                    next_state = SEND_SIGNALS;
            SEND_SIGNALS:
                if (!empty && !blanking)
                    next_state = IDLE;
                else if (!empty)
                    next_state = EMPTY_LOW;
                else if (!blanking)
                    next_state = BLANKING_LOW;
                else
                    next_state = NEITHER_LOW;
            NEITHER_LOW:
                if (!empty && !blanking)
                    next_state = IDLE;
                else if (!empty)
                    next_state = EMPTY_LOW;
                else if (!blanking)
                    next_state = BLANKING_LOW;
            EMPTY_LOW:
                if (!blanking)
                    next_state = IDLE;
            BLANKING_LOW:
                if (!empty)
                    next_state = IDLE;
        endcase
    end

    assign next_frame = (state == SEND_SIGNALS);
    assign switch_buffer = (state == SEND_SIGNALS);

endmodule

`timescale 1ns / 1ps
module delay(reset, clock, in, out);
    input reset;
    input clock;

    parameter DELAY = 2;

    input in;
    output out;

    reg [DELAY-1:0] data;

    assign out = data[DELAY-1];

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



```
            data <= {DELAY{1'b0}};
        else
            data <= {data[DELAY - 2: 0], in};
    end

endmodule

//  VGA Module from Problem Set 3
module vga (reset, pixel_clock, sync_b,
            blank_b, hsync, vsync, pixel_count,
            line_count, enable, blanking);

    input reset; // system reset
    input pixel_clock; // 31.5 MHz pixel clock
    output sync_b; // hardwired to Vdd
    output blank_b; // composite blank
    output hsync; // horizontal sync
    output vsync; // vertical sync
    output [9:0] pixel_count; // number of the current pixel
    output [9:0] line_count; // number of the current line
    output enable;  // 1-clock enable once per screen refresh
    output blanking;

//  // 640x480 75Hz parameters
//  parameter PIXELS = 800;
//  parameter LINES = 525;
//  parameter HACTIVE_VIDEO = 640;
//  parameter HFRONT_PORCH = 16;
//  parameter HSYNC_PERIOD = 96;
//  parameter HBACK_PORCH = 48;
//  parameter VACTIVE_VIDEO = 480;
//  parameter VFRONT_PORCH = 11;
//  parameter VSYNC_PERIOD = 2;
//  parameter VBACK_PORCH = 32;
//    640x480, 75Hz  31.500 640 16  96  48  480 11  2   32
//    640x480, 60Hz  25.175 640 16  96  48  480 11  2   31
    // 640x480 60Hz parameters
    parameter PIXELS = 800;
    parameter LINES = 524;
    parameter HACTIVE_VIDEO = 640;
    parameter HFRONT_PORCH = 16;
    parameter HSYNC_PERIOD = 96;
    parameter HBACK_PORCH = 48;
    parameter VACTIVE_VIDEO = 480;
    parameter VFRONT_PORCH = 11;
    parameter VSYNC_PERIOD = 2;
    parameter VBACK_PORCH = 31;


    // current pixel count
    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;
    reg enable = 1'b0;
    reg blanking;
    wire sync_b; // connected to Vdd

    wire pixel_clock;
    wire [9:0] next_pixel_count;
    wire [9:0] next_line_count;
```

internal.v

```
    wire next_enable;

    always @ (posedge pixel_clock)
        begin
        if (reset)
            begin
                pixel_count <= 10'b0;
                line_count <= 10'b0;
                hsync <= 1'b1;
                vsync <= 1'b1;
                blank_b <= 1'b1;
                enable <= 1'b0;
            end
        else
            begin
                pixel_count <= next_pixel_count;
                line_count <= next_line_count;
                hsync <=
                    (next_pixel_count < HACTIVE_VIDEO + HFRONT_PORCH) |
                    (next_pixel_count >= HACTIVE_VIDEO+HFRONT_PORCH+
                    HSYNC_PERIOD);
                vsync <=
                    (next_line_count < VACTIVE_VIDEO+VFRONT_PORCH) |
                    (next_line_count >= VACTIVE_VIDEO+VFRONT_PORCH+
                VSYNC_PERIOD);
                // this is the and of hblank and vblank
                blank_b <=
                    (next_pixel_count < HACTIVE_VIDEO) &
                    (next_line_count < VACTIVE_VIDEO);
                enable <= next_enable;
                blanking <=
                    (next_line_count > VACTIVE_VIDEO &&
                    next_line_count < LINES);
            end
        end

    // next state is computed with combinational logic
    assign next_pixel_count =
        (pixel_count == PIXELS-1) ?
         10'h000 : pixel_count + 1'b1;
    assign next_line_count =
        (pixel_count == PIXELS-1) ?
         (line_count == LINES-1) ? 10'h000 :
         line_count + 1'b1 : line_count;
    // can hardwire composite sync to Vdd.
    assign sync_b = 1'b1;

    assign next_enable =
        (pixel_count == 1)
        && (line_count == VACTIVE_VIDEO - 2);

endmodule
```


screen_buffer.v

```
`timescale 1ns / 1ps
`include "defines.v"
module screen_buffer(
    reset, clock,

    switch_buffer,

    write_pixel_count, write_line_count,
    write_z, write_rgb, write_noop, write_next,

    read_pixel_count, read_line_count, read_rgb,

    ram0_data, ram0_address, ram0_we_b, // ram 0 buss
    ram1_data, ram1_address, ram1_we_b // ram 1 buss
    );
    input reset;
    input clock;

    input switch_buffer;

    input [9:0] write_pixel_count;
    input [9:0] write_line_count;
    input [11:0] write_z;
    input [23:0] write_rgb;
    input write_noop;
    output write_next;

    input [9:0] read_pixel_count;
    input [9:0] read_line_count;
    output [23:0] read_rgb;

    inout [35:0] ram0_data;
    output [18:0] ram0_address;
    output ram0_we_b;

    inout [35:0] ram1_data;
    output [18:0] ram1_address;
    output ram1_we_b;

    // wiring
    wire [35:0] read_data;
    wire [18:0] read_address;
    wire read_we_b;

    wire [18:0] write_address;
    wire write_we_b;
    wire hold;

    // state
    reg ram_select;

    always @(posedge clock)
        if (reset)
            ram_select <= 0;
        else
            ram_select <= switch_buffer ? !ram_select : ram_select;

    assign {ram0_we_b, ram1_we_b} = ram_select ?
                                    {read_we_b, write_we_b}:
                                    {write_we_b, read_we_b};
    assign {ram0_address, ram1_address} = ram_select ?
                                        {read_address, write_address}:
                                        {write_address, read_address};
```

```

    assign read_data = ram_select ? ram0_data: ram1_data;

    zbt_reader reader(reset, clock, read_pixel_count, read_line_count, read_rgb,
                      read_we_b,
                      read_address,
                      read_data);


    zbt_out out(reset, clock, write_pixel_count, write_line_count,
                write_z, write_rgb, write_noop,
                write_we_b, write_address, ram_select,
                ram0_data, ram1_data, switch_buffer, write_next);
endmodule

module zbt_out(reset, clock, pixel_count, line_count, pixel_z, rgb, noop,
               ram_we_b, ram_address, ram_select,
               ram0_data, ram1_data, switch_buffer, next);


    parameter DEFAULT_VALUE = {12'b111111111111, 24'b0};
    parameter MAX_ADDRESS = `SCREEN_HEIGHT * `SCREEN_WIDTH;

    input reset;
    input clock;
    input [9:0] pixel_count;
    input [9:0] line_count;
    input [23:0] rgb;
    input [11:0] pixel_z;
    input noop;
    output next;

    output ram_we_b;    // physical line to ram we_b
    output [18:0] ram_address;  // physical line to ram address
    input ram_select;

    inout  [35:0] ram0_data;    // physical line to ram data
    inout  [35:0] ram1_data; // physical line to ram data

    input switch_buffer;

    // buffer the outputs
    reg ram_we_b;
    reg [18:0] ram_address;
    reg [35:0] ram_data_out;
    reg next;

    // internal state
    reg oe;
    reg [1:0] state;

    localparam CLEARING = 0;
    localparam WRITING = 1;
    localparam READING = 2;

    // additional input
    wire [35:0] ram_data_in = (ram_select) ? ram1_data: ram0_data;
    assign ram1_data = (oe && ram_select) ? ram_data_out : {36{1'bZ}};
    assign ram0_data = (oe && !ram_select) ? ram_data_out : {36{1'bZ}};

    // shift registers
    reg [35:0] ram_data_in0, ram_data_in1, ram_data_in2;
    reg [18:0] ram_address0, ram_address1, ram_address2;
```



```
    reg [23:0] rgb0, rgb1, rgb2;
    reg [11:0] pixel_z0, pixel_z1, pixel_z2;
    reg noop0, noop1, noop2;

    always @ (posedge clock) begin
        {ram_data_in0, ram_data_in1, ram_data_in2}
            <= {ram_data_in, ram_data_in0, ram_data_in1};
        {ram_address0, ram_address1, ram_address2}
            <= {ram_address, ram_address0, ram_address1};
        {rgb0, rgb1, rgb2} <= {rgb, rgb0, rgb1};
        {pixel_z0, pixel_z1, pixel_z2} <= {pixel_z, pixel_z0, pixel_z1};
        {noop0, noop1, noop2} <= {noop, noop0, noop1};
    end

    // next state wires
    reg next_we;
    reg [18:0] next_address;
    reg [35:0] next_data_out;
    reg next_oe;
    reg [1:0] next_state;

    always @ (posedge clock)
        if (reset) begin
            ram_we_b <= 1'b1;
            ram_address <= 19'b0;
            ram_data_out <= 36'b0;
            oe <= 1'b0;
            state <= READING;
        end else begin
            ram_we_b <= !next_we;
            ram_address <= next_address;
            ram_data_out <= next_data_out;
            oe <= next_oe;
            state <= next_state;
        end

    always @ (state, ram_address, switch_buffer, line_count, pixel_count, oe,
             ram_we_b, ram_data_out, noop, pixel_z2, ram_data_in0, rgb2,
             noop1, ram_data_in) begin
        next_state = state;
        next_address = ram_address;
        next_oe = oe;
        next_we = !ram_we_b;
        next_data_out = ram_data_out;
        next = 1'b0;
        case (state)
            CLEARING:
            begin
                next = 1'b0;
                if (ram_address == MAX_ADDRESS+1) begin
                    next_state = READING;
                    next_data_out = DEFAULT_VALUE;
                    next_oe = 1'b1;
                    next_we = 1'b0;
                end else begin
                    next_address = ram_address + 1;
                    next_data_out = DEFAULT_VALUE;
                    next_oe = 1'b1;
                    next_we = 1'b1;
                end
            end
            WRITING:
            begin
```

```
                next = 1'b1;
                if (switch_buffer) begin
                    next = 1'b0;
                    next_state = CLEARING;
                    next_address = 19'b0;
                    next_oe = 1'b0;
                    next_we = 1'b1;
                end else begin
                    next_address =
                        (line_count>=`SCREEN_HEIGHT ||
                         pixel_count>=`SCREEN_WIDTH)
                         ? -1 : line_count * `SCREEN_WIDTH + pixel_count;
                    next_oe = 1'b0;
                    next_we = 1'b0;
                    next_state = READING;
                end
            end
            READING:
            begin
                next = 1'b0;
                if (switch_buffer) begin
                    next_state = CLEARING;
                    next_address = 19'b0;
                    next_oe = 1'b0;
                    next_we = 1'b1;
                end else begin
                    next_we = !noop0;//!noop;//1'b1;
                    next_oe = !noop2;//!noop1;//1'b1;
                    next_data_out = (pixel_z2 > ram_data_in[35:24])
                        ? ram_data_in : {pixel_z2, rgb2};
                    next_state = WRITING;
                end
            end
        endcase
    end
endmodule

module zbt_reader(
    reset, clock, pixel_count, line_count, read_rgb,
    ram_we_b, ram_address, ram_data);
    input clock;            // system clock
    input reset;
    input [9:0] pixel_count;
    input [9:0] line_count;
    output [23:0] read_rgb; // data read from memory

    output ram_we_b;    // physical line to ram we_b
    output [18:0] ram_address;  // physical line to ram address
    input  [35:0] ram_data; // physical line to ram data

    assign ram_we_b = 1'b1;
    reg [18:0] ram_address;
    reg [23:0] read_rgb, read_rgb0, read_rgb1;

    wire [18:0] next_address =
        (line_count>=`SCREEN_HEIGHT ||
         pixel_count>=`SCREEN_WIDTH)
         ? -1 : line_count * `SCREEN_WIDTH + pixel_count;
    always @ (posedge clock)
        if (reset) begin
            ram_address <= 36'b0;
            read_rgb <= 36'b0;
            read_rgb0 <= 36'b0;
```



```
            read_rgb1 <= 36'b0;
        end else begin
            ram_address <= next_address;
            {read_rgb, read_rgb0, read_rgb1}
                <= {read_rgb0, read_rgb1, ram_data[23:0]};
        end
endmodule
```

```
`timescale 1ns / 1ps
`include "defines.v"
module triangle_shader(
    reset, clock, triangle_data, triangle_empty,
    triangle_noop, triangle_next,
    pixel_data, pixel_empty, pixel_noop, pixel_next);
    // input
    input reset;
    input clock;
    input [`TRIANGLE_BITS-1:0] triangle_data;
    input triangle_empty;
    input triangle_noop;
    output triangle_next;

    // output
    output [`PIXEL_BITS-1:0] pixel_data;
    output pixel_empty;
    output pixel_noop;
    input pixel_next;

    // register some outputs
    reg [`PIXEL_BITS-1:0] pixel_data;
    reg pixel_empty;
    reg triangle_next;

    // register state
    reg [1:0] state;
    localparam RHOMBUS_A = 0;
    localparam RHOMBUS_B = 1;
    localparam NEXT = 2;

    // inputs to rhombus shader
    reg [`RHOMBUS_BITS-1:0] rhombus_data;
    reg rhombus_noop;
    wire rhombus_next;

    // next state values
    reg [1:0] next_state;
    wire [`PIXEL_BITS-1:0] next_pixel_data;
    wire [`POINT_BITS-1:0] next_point_data;
    reg next_pixel_empty;
    reg next_rhombus_noop;
    reg [`RHOMBUS_BITS-1:0] next_rhombus_data;

    rhombus_shader rs (reset, clock, rhombus_data, rhombus_noop, rhombus_next,
                    next_point_data, pixel_noop, pixel_next);

    // split up rhombus_painter inputs
    wire [`RHOMBUS_BITS-1:0] rhombus_a, rhombus_b;
    wire [`COORD_BITS-1:0] a_min_x, a_min_y, a_max_x, a_max_y,
                            b_min_x, b_min_y, b_max_x, b_max_y,
                            a_z, b_z;
    wire [`LINE_BITS-1:0] a_left_line, a_right_line, b_left_line, b_right_line;
    assign rhombus_a = {a_left_line, a_right_line,
                        a_min_x, a_min_y, a_max_x, a_max_y, a_z};
    assign rhombus_b = {b_left_line, b_right_line,
                        b_min_x, b_min_y, b_max_x, b_max_y, b_z};

    // split up triangle_data inputs
    wire [`POINT_BITS-1:0] top_point, left_point,
                            right_point, bottom_point, mid_point;
    wire [`COORD_BITS-1:0]  top_x, top_y, top_z,
                            left_x, left_y, left_z, right_x, right_y, right_z,
```



```
                            bottom_x, bottom_y, bottom_z, mid_x, mid_y, mid_z;
    wire [`COLOR_BITS-1:0] rgb;
    assign {top_point, left_point, right_point, rgb} = triangle_data;
    assign {top_x, top_y, top_z} = top_point;
    assign {left_x, left_y, left_z} = left_point;
    assign {right_x, right_y, right_z} = right_point;
    reg [`COLOR_BITS-1:0] rgb0;

    // find mid/bottom point
    wire left_bellow_right;
    signed_lt #(`COORD_BITS) slt (left_y, right_y, left_bellow_right);
    assign {bottom_point, mid_point} = left_bellow_right ?
        {left_point, right_point} : {right_point, left_point};
    assign {bottom_x, bottom_y, bottom_z} = bottom_point;
    assign {mid_x, mid_y, mid_z} = mid_point;

    //assign min/max
    wire [`COORD_BITS-1:0] min_x, max_x;
    signed_outer3 #(`COORD_BITS) so3 (left_x, top_x, right_x, min_x, max_x);

    assign a_min_x = min_x;
    assign b_min_x = min_x;
    assign a_max_x = max_x;
    assign b_max_x = max_x;

    assign a_max_y = top_y;
    assign a_min_y = mid_y;
    assign b_max_y = mid_y;
    assign b_min_y = bottom_y;

    // calculate average z (kind of)
    wire [`COORD_BITS+1:0] z_sum;
    wire [`COORD_BITS-1:0] average_z;
    signed_add3 #(`COORD_BITS) sa3 (top_z, left_z, right_z, z_sum);
    assign average_z = z_sum[`COORD_BITS+1:2];
    assign a_z = average_z;
    assign b_z = average_z;

    // assign lines
    assign a_left_line = {top_x, top_y, left_x, left_y};
    assign a_right_line = {top_x, top_y, right_x, right_y};
    assign b_left_line = left_bellow_right ?
        {top_x, top_y, left_x, left_y} : {left_x, left_y, right_x, right_y};
    assign b_right_line = left_bellow_right ?
        {right_x, right_y, left_x, left_y} : {top_x, top_y, right_x, right_y};

    //
    always @ (posedge clock)
        if (reset) begin
            state <= NEXT;
            rhombus_data <= {(`RHOMBUS_BITS){1'b0}};
            rhombus_noop <= 1'b1;
            pixel_empty <= 1'b0;
            pixel_data <= {(`PIXEL_BITS){1'b0}};
            rgb0 <= 24'b0;
        end else begin
            state <= next_state;
            rhombus_data <= next_rhombus_data;
            rhombus_noop <= next_rhombus_noop;
            pixel_empty <= next_pixel_empty;
            pixel_data <= next_pixel_data;
            rgb0 <= rgb;
        end
```

```
    // calculate next values
    point_to_pixel ptp (next_point_data, rgb0, next_pixel_data);
    always @ (state, rhombus_noop, rhombus_data, rhombus_a, rhombus_b,
              rhombus_next, triangle_empty, triangle_noop) begin
        next_state = state;
        next_pixel_empty = 1'b0;
        next_rhombus_noop = rhombus_noop;
        next_rhombus_data = rhombus_data;
        triangle_next = 1'b0;

        case (state)
            RHOMBUS_A:
                if (rhombus_next) begin
                    next_state = RHOMBUS_B;
                    next_rhombus_data = rhombus_b;
                end
            RHOMBUS_B:
                if (rhombus_next) begin
                    next_state = NEXT;
                    next_rhombus_noop = 1'b1;
                    triangle_next = 1'b1;
                end
            NEXT:
            begin
                if (triangle_empty) begin
                    triangle_next = 1'b1;
                    next_pixel_empty = 1'b1;
                end else if (triangle_noop) begin
                    triangle_next = 1'b1;
                end else if (rhombus_next) begin
                    next_state = RHOMBUS_A;
                    next_rhombus_data = rhombus_a;
                    next_rhombus_noop = 1'b0;
                end
            end
        endcase
    end
endmodule

module point_to_pixel(point_data, rgb, pixel_data/*, pixel_noop*/);
    input [`POINT_BITS-1:0] point_data;
    input [`COLOR_BITS-1:0] rgb;
    output [`PIXEL_BITS-1:0] pixel_data;
    //output pixel_noop;

    wire [`COORD_BITS-1:0] unsigned_x, unsigned_z;
    wire [`COORD_BITS-1:0] point_x, point_y, point_z;
    wire [`COORD_BITS:0] neg_y, unsigned_y;

    assign {point_x, point_y, point_z} = point_data;

    signed_negate #(`COORD_BITS) sn (point_y, neg_y);

    signed_to_unsigned #(`COORD_BITS, (`SCREEN_WIDTH/2))
        stux (point_x, unsigned_x);
    signed_to_unsigned #(`COORD_BITS+1, (`SCREEN_HEIGHT/2))
        stuy (neg_y, unsigned_y);
    signed_to_unsigned #(`COORD_BITS, {1'b1, {`COORD_BITS-1{1'b0}}})
        stuz (point_z, unsigned_z);


    assign pixel_data = {unsigned_x[9:0], unsigned_y[9:0],
```



```
                         unsigned_z[`COORD_BITS-1:`COORD_BITS-12], rgb};
    //assign pixel_noop = (unsigned_x[`COORD_BITS-1:10] != 0) ||
    //                    (unsigned_y[`COORD_BITS-1:10] != 0);
endmodule

`include "defines.v"
module rhombus_shader(
    reset, clock, rhombus_data, rhombus_noop, rhombus_next,
    next_point_data, point_noop, point_next);
    input reset;
    input clock;

    input [`RHOMBUS_BITS-1:0] rhombus_data;
    input rhombus_noop;
    output rhombus_next;

    output [`POINT_BITS-1:0] next_point_data;
    output point_noop;
    input point_next;

    // split up rhombus data
    wire [`LINE_BITS-1:0] left_line, right_line;
    wire [`COORD_BITS-1:0] min_x, min_y, max_x, max_y, rhombus_z;
    assign {left_line, right_line,
            min_x, min_y,
            max_x, max_y, rhombus_z} = rhombus_data;

    // register point data output
    reg [`COORD_BITS-1:0] point_x, point_y, point_z;
    reg point_noop;

    // state
    reg state;
    localparam IDLE = 0;
    localparam PAINTING = 1;

    // next state variables
    reg [`COORD_BITS-1:0] next_point_x, next_point_y, next_point_z;
    assign next_point_data = {next_point_x, next_point_y, next_point_z};
    reg next_point_noop, next_state;
    reg rhombus_next;

    wire [`COORD_BITS:0] inc_x, inc_y;
    signed_inc #(`COORD_BITS) six(point_x, inc_x);
    signed_inc #(`COORD_BITS) siy(point_y, inc_y);

    always @ (posedge clock)
        if (reset) begin
            point_x = {(`POINT_BITS){1'b0}};
            point_y = {(`POINT_BITS){1'b0}};
            point_z = {(`POINT_BITS){1'b0}};
            state = IDLE;
            point_noop = 1'b1;
        end else begin
            point_x = next_point_x;
            point_y = next_point_y;
            point_z = next_point_z;
            state = next_state;
            point_noop = next_point_noop;
        end

    wire before_left_line;
    wire after_right_line;
```

```

    always @ (state, rhombus_noop, point_next, point_y, max_y, point_x,
              min_x, min_y, rhombus_z, max_x, max_y, point_z, point_noop,
              before_left_line, after_right_line, inc_y, inc_x) begin
        next_state = state;
        next_point_x = point_x;
        next_point_y = point_y;
        next_point_z = point_z;
        next_point_noop = point_noop;
        rhombus_next = 1'b0;
        case (state)
            IDLE:
                if (rhombus_noop) begin
                    rhombus_next = 1'b1;
                    next_point_noop = 1'b1;
                end else begin
                    next_state = PAINTING;
                    next_point_x = min_x;
                    next_point_y = min_y;
                    next_point_z = rhombus_z;
                    next_point_noop = before_left_line || after_right_line;
                end
            PAINTING:
                if (point_next) begin
                    if (point_y == max_y) begin
                        next_state = IDLE;
                        rhombus_next = 1'b1;
                        next_point_noop = 1'b1;
                    end else if (point_x == max_x) begin
                        next_point_x = min_x;
                        next_point_y = inc_y[`COORD_BITS-1:0];
                        next_point_noop = before_left_line || after_right_line;
                    end else begin
                        next_point_x = inc_x[`COORD_BITS-1:0];
                        next_point_noop = before_left_line || after_right_line;
                    end
                end
        endcase
    end

    //point_left_of_line
    wire before_right_line;

    point_left_of_line plol1
        (next_point_x, next_point_y, left_line, before_left_line);
    point_left_of_line plol2
        (next_point_x, next_point_y, right_line, before_right_line);

    assign after_right_line = ~before_right_line;
endmodule

module point_left_of_line(p_x, p_y, line, out);
    input [`COORD_BITS-1:0] p_x, p_y;
    input [`LINE_BITS-1:0] line;
    output out;

    wire [`COORD_BITS-1:0] a_x, a_y, b_x, b_y;
    assign {a_x, a_y, b_x, b_y} = line;


    wire [`COORD_BITS:0] pxmax, aymby, pymay, axmbx;

    signed_subtract #(`COORD_BITS) ss1 (p_x, a_x, pxmax);
```



```
    signed_subtract #(`COORD_BITS) ss2 (a_y, b_y, aymby);
    signed_subtract #(`COORD_BITS) ss3 (p_y, a_y, pymay);
    signed_subtract #(`COORD_BITS) ss4 (a_x, b_x, axmbx);

    wire [2*(`COORD_BITS)+1:0] left, right;
    signed_multiply #(`COORD_BITS+1,`COORD_BITS+1) sm1 (pxmax, aymby, left);
    signed_multiply #(`COORD_BITS+1,`COORD_BITS+1) sm2 (pymay, axmbx, right);

    wire left_lt_right, ay_lt_by;
    signed_lt #(2*(`COORD_BITS)+2) slt1 (left, right, left_lt_right);
    signed_lt #(`COORD_BITS) slt2 (a_y, b_y, ay_lt_by);

    assign out = left_lt_right ^ ay_lt_by;
endmodule

```

```
`include "defines.v"
module triangle_pipeline(
    reset, clock,
    in_triangle, in_noop, in_empty, in_next,
    out_triangle, out_noop, out_empty, out_next,
    translation_vector, rotation_matrix, light_vector, zoom,
    rotor_matrix, ball_vector, paddle_vector);
    input reset;
    input clock;

    // incomming triangles
    input [`TRIANGLE_BITS+`NORMAL_BITS-1:0] in_triangle;
    input in_empty;
    input in_noop;
    output in_next;

    // outgoing triangles
    output [`TRIANGLE_BITS-1:0] out_triangle;
    output out_empty;
    output out_noop;
    input out_next;

    // control signals
    input [3*`COORD_BITS-1:0] translation_vector, ball_vector,
                              paddle_vector, light_vector;
    input [9*`TRIG_OUT_BITS-1:0] rotation_matrix, rotor_matrix;
    input [`COORD_BITS-1:0] zoom;

    // next propogation wiring
    assign in_next = out_next;

    // internal wiring

    wire [`NORMAL_BITS-1:0] next_normal0, next_normal1, next_normal2, next_normal3,
                            next_normal4, next_normal5, next_normal6,
                            normal0, normal1, normal2, normal3, normal4,
                            normal5, normal6;

    wire [`TRIANGLE_BITS-1:0]
        next_data0, next_data1, next_data2, next_data3, next_data4, next_data5,
        next_data6, next_data7, next_data8, next_data9, next_data10, next_data11,
        next_data12, next_data13,
        data0, data1, data2,  data3, data4, data5, data6, data7, data8, data9,
        data10, data11, data12, data13;
    wire next_noop0, next_noop1, next_noop2, next_noop3, next_noop4, next_noop5,
        next_noop6, next_noop7, next_noop8, next_noop9, next_noop10, next_noop11,
        next_noop12, next_noop13,
        noop0, noop1, noop2, noop3, noop4, noop5, noop6, noop7, noop8, noop9,
        noop10, noop11, noop12, noop13,
        empty0, empty1, empty2, empty3, empty4, empty5, empty6, empty7, empty8,
        empty9, empty10, empty11, empty12, empty13;

    assign {data0, normal0} = in_triangle;
    assign noop0 = in_noop;
    assign empty0 = in_empty;

    pipeline_register #(`TRIANGLE_BITS+`NORMAL_BITS) pr0
        (reset, clock, out_next, {data0,normal0}, noop0, empty0,
            {data1, normal1}, noop1, empty1);

    wire [`TRIANGLE_BITS-1:0] optdata2;
    wire [`NORMAL_BITS-1:0] optnormal2;
    wire optnoop2, optempty2;
```



```
    rotation_block b1
        (data1, normal1, next_data1, next_normal1, next_noop1, rotor_matrix);
    pipeline_register #(`TRIANGLE_BITS+`NORMAL_BITS) pr1
        (reset, clock, out_next, {data1,normal1}, noop1, empty1,
            {data2, normal2}, noop2, empty2);
    pipeline_register #(`TRIANGLE_BITS+`NORMAL_BITS) pr1_opt
        (reset, clock, out_next, {next_data1,next_normal1}, next_noop1, empty1,
            {optdata2, optnormal2}, optnoop2, optempty2);

    opt_block #(`ROTOR_COLOR) b2
        (optdata2, optnormal2, optnoop2, data2, normal2,
            next_data2, next_normal2, next_noop2);
    pipeline_register #(`TRIANGLE_BITS+`NORMAL_BITS) pr2
        (reset, clock, out_next, {next_data2,next_normal2}, next_noop2||noop2, empty2,
            {data3, normal3}, noop3, empty3);

    opt_translation_block #(`BALL_COLOR) b3 (data3, next_data3, next_noop3, ball_vector);

    pipeline_register #(`TRIANGLE_BITS+`NORMAL_BITS) pr3
        (reset, clock, out_next, {next_data3, normal3}, next_noop3||noop3, empty3,
            {data4, normal4}, noop4, empty4);

    opt_translation_block #(`PADDLE_COLOR) b4
        (data4, next_data4, next_noop4, paddle_vector);
    pipeline_register #(`TRIANGLE_BITS+`NORMAL_BITS) pr4
        (reset, clock, out_next, {next_data4, normal4}, next_noop4||noop4, empty4,
            {data5, normal5}, noop5, empty5);

    rotation_block b5
        (data5, normal5, next_data5, next_normal5, next_noop5, rotation_matrix);
    pipeline_register #(`TRIANGLE_BITS+`NORMAL_BITS) pr5
        (reset, clock, out_next, {next_data5,next_normal5}, next_noop5||noop5, empty5,
            {data6, normal6}, noop6, empty6);

    wire [9:0] next_change, change;
    light_block0 b6 (data6, normal6, next_data6, next_change, next_noop6, light_vector);
    pipeline_register #(`TRIANGLE_BITS+10) pr6
        (reset, clock, out_next, {next_data6, next_change}, next_noop6||noop6, empty6,
            {data7, change}, noop7, empty7);

    light_block1 b7 (data7, change, next_data7, next_noop7);
    pipeline_register #(`TRIANGLE_BITS) pr7
        (reset, clock, out_next, next_data7, next_noop7||noop7, empty7,
            data8, noop8, empty8);

    normal_block b8 (data8, next_data8, next_noop8);
    pipeline_register #(`TRIANGLE_BITS) pr8
        (reset, clock, out_next, next_data8, next_noop8||noop8, empty8,
            data9, noop9, empty9);

    reorder_block b9 (data9, next_data9, next_noop9);
    pipeline_register #(`TRIANGLE_BITS) pr9
        (reset, clock, out_next, next_data9, next_noop9||noop9, empty9,
            data10, noop10, empty10);

    zoom_block b10 (data10, next_data10, next_noop10, zoom);
    pipeline_register #(`TRIANGLE_BITS) pr10
        (reset, clock, out_next, next_data10, next_noop10||noop10, empty10,
            data11, noop11, empty11);

    translation_block b11 (data11, next_data11, next_noop11, translation_vector);
    pipeline_register #(`TRIANGLE_BITS) pr11
        (reset, clock, out_next, next_data11, next_noop11||noop11, empty11,
```

```
            data12, noop12, empty12);

    wire [4*(`COORD_BITS)-1:0] next_extrema, extrema;
    keep_visible_block0 b12 (data12, next_data12, next_extrema, next_noop12);
    pipeline_register #(`TRIANGLE_BITS+(4*(`COORD_BITS))) pr12
        (reset, clock, out_next, {next_data12, next_extrema},
            next_noop12||noop12, empty12,
            {data13, extrema}, noop13, empty13);

    keep_visible_block1 b13 (data13, extrema, next_data13, next_noop13);
    pipeline_register #(`TRIANGLE_BITS) pr13
        (reset, clock, out_next, next_data13, next_noop13||noop13, empty13,
            out_triangle, out_noop, out_empty);

endmodule

module keep_visible_block0(in_triangle, out_triangle, out_extrema, out_noop);
    input [`TRIANGLE_BITS-1:0] in_triangle;

    // outgoing triangles
    output [`TRIANGLE_BITS-1:0] out_triangle;
    output [4*(`COORD_BITS)-1:0] out_extrema;
    output out_noop;

    // split up inputs
    wire [`POINT_BITS-1:0] in_a, in_b, in_c;
    wire [`COORD_BITS-1:0] a_x, a_y, a_z, b_x, b_y, b_z, c_x, c_y, c_z;
    wire [`COLOR_BITS-1:0] in_rgb;

    assign {in_a, in_b, in_c, in_rgb} = in_triangle;
    assign {a_x, a_y, a_z} = in_a;
    assign {b_x, b_y, b_z} = in_b;
    assign {c_x, c_y, c_z} = in_c;
    assign out_triangle = in_triangle;

    // compare the y coordinates
    wire [`COORD_BITS-1:0] min_x, max_x, min_y, max_y;

    signed_outer3 #(`COORD_BITS) sox (a_x, b_x, c_x, min_x, max_x);
    signed_outer3 #(`COORD_BITS) soy (a_y, b_y, c_y, min_y, max_y);

    assign out_extrema = {min_x, max_x, min_y, max_y};

    assign out_noop = 0;
endmodule

module keep_visible_block1(in_triangle, in_extrema, out_triangle, out_noop);
    input [`TRIANGLE_BITS-1:0] in_triangle;
    input [4*(`COORD_BITS)-1:0] in_extrema;

    // outgoing triangles
    output [`TRIANGLE_BITS-1:0] out_triangle;
    output out_noop;

    // split up inputs
    assign out_triangle = in_triangle;

    // compare the y coordinates
    wire [`COORD_BITS-1:0] min_x, max_x, min_y, max_y;
    assign {min_x, max_x, min_y, max_y} = in_extrema;

    wire slt0, slt1, slt2, slt3;
    signed_lt #(`COORD_BITS) slt_0 (max_x, -(`SCREEN_WIDTH/2), slt0);
```



```
    signed_lt #(`COORD_BITS) slt_1 ((`SCREEN_WIDTH/2), min_x, slt1);
    signed_lt #(`COORD_BITS) slt_2 (max_y, -(`SCREEN_HEIGHT/2), slt2);
    signed_lt #(`COORD_BITS) slt_3 ((`SCREEN_HEIGHT/2), min_y, slt3);

    // assing noop output
    assign out_noop = 0;//slt0||slt1||slt2||slt3;
endmodule

module light_block0(
    in_triangle, in_normal, out_triangle, out_change, out_noop, light_vector);
    input [`TRIANGLE_BITS-1:0] in_triangle;
    input [`NORMAL_BITS-1:0] in_normal;

    // outgoing triangles
    output [`TRIANGLE_BITS-1:0] out_triangle;
    output [9:0] out_change;
    output out_noop;

    input [3*`COORD_BITS-1:0] light_vector;

    // split up inputs and outputs
    assign out_triangle = in_triangle;
    assign out_noop = 0;

    wire [(2*`COORD_BITS+2)-1:0] correlation;

    dot_product #(`COORD_BITS, `COORD_BITS) dp1 (in_normal, light_vector, correlation);

    assign out_change = correlation[2*`COORD_BITS-1:2*`COORD_BITS-10];
endmodule

module light_block1(in_triangle, in_change, out_triangle, out_noop);
    input [`TRIANGLE_BITS-1:0] in_triangle;
    input [9:0] in_change;

    // outgoing triangles
    output [`TRIANGLE_BITS-1:0] out_triangle;
    output out_noop;

    // split up inputs and outputs
    wire [7:0] out_r, out_g, out_b, in_r, in_g, in_b;
    wire [3*`POINT_BITS-1:0] in_points;
    assign {in_points, in_r, in_g, in_b} = in_triangle;
    assign out_triangle = {in_points, out_r, out_g, out_b};
    assign out_noop = 0;

    wire [10:0] temp_r, temp_g, temp_b;

    signed_add #(10) sar ({2'b0, in_r}, in_change, temp_r);
    signed_add #(10) sag ({2'b0, in_g}, in_change, temp_g);
    signed_add #(10) sab ({2'b0, in_b}, in_change, temp_b);

    wire [10:0] bound_r, bound_g, bound_b;

    signed_bound #(11, 255, 0) sbr (temp_r, bound_r);
    signed_bound #(11, 255, 0) sbg (temp_g, bound_g);
    signed_bound #(11, 255, 0) sbb (temp_b, bound_b);

    assign out_r = bound_r[7:0];
    assign out_g = bound_g[7:0];
    assign out_b = bound_b[7:0];
endmodule
```

```
module normal_block(in_triangle, out_triangle, out_noop);
    input [`TRIANGLE_BITS-1:0] in_triangle;

    // outgoing triangles
    output [`TRIANGLE_BITS-1:0] out_triangle;
    //output [3*(2*`COORD_BITS+3)-1:0] out_normal;

    output out_noop;

    // split up inputs and outputs
    wire [`POINT_BITS-1:0] in_a, in_b, in_c;
    wire [`COLOR_BITS-1:0] in_rgb;

    assign {in_a, in_b, in_c, in_rgb} = in_triangle;
    assign out_triangle = in_triangle;


    wire [3*(`COORD_BITS+1)-1:0] a_to_b, a_to_c;
    vector_subtract #(`COORD_BITS) vs_1 (in_b, in_a, a_to_b);
    vector_subtract #(`COORD_BITS) vs_2 (in_c, in_a, a_to_c);

    wire [(2*`COORD_BITS+3)-1:0] n_x, n_y, n_z;
    cross_product #(`COORD_BITS+1, `COORD_BITS+1) cp (a_to_b, a_to_c, {n_x, n_y, n_z});

    assign out_noop = n_z[(2*`COORD_BITS+3)-1];
    //assign out_normal = {n_x, n_y, n_z};
endmodule

module opt_block(
    opt_triangle, opt_normal, opt_noop, in_triangle, in_normal,
    out_triangle, out_normal, out_noop);
    parameter COLOR = 24'b1;

    input [`TRIANGLE_BITS-1:0] in_triangle, opt_triangle;
    input [`NORMAL_BITS-1:0] in_normal, opt_normal;
    input opt_noop;

    // outgoing triangles
    output [`TRIANGLE_BITS-1:0] out_triangle;
    output [`NORMAL_BITS-1:0] out_normal;
    output out_noop;

    assign {out_triangle, out_normal, out_noop} =
           ((in_triangle[23:0] == COLOR) ||
            (in_triangle[23:0] == `MIT_RED) ||
            (in_triangle[23:0] == `MIT_GRAY))?
           {opt_triangle, opt_normal, opt_noop}:
           {in_triangle, in_normal, 1'b0};
endmodule

module opt_translation_block(in_triangle, out_triangle, out_noop, translation_vector);
    parameter COLOR = 24'b1;

    input [`TRIANGLE_BITS-1:0] in_triangle;

    // outgoing triangles
    output [`TRIANGLE_BITS-1:0] out_triangle;
    output out_noop;

    input [3*`COORD_BITS-1:0] translation_vector;

    wire [`TRIANGLE_BITS-1:0] out_triangle0;
    wire out_noop0;
```



```

    translation_block tb (in_triangle, out_triangle0, out_noop0, translation_vector);

    assign {out_triangle, out_noop} =
       in_triangle[23:0] == COLOR ?
           {out_triangle0, out_noop0}:
           {in_triangle, 1'b0};
endmodule

module pipeline_register(
    reset, clock, next, in_data, in_noop, in_empty, out_data, out_noop, out_empty);

    parameter WIDTH = 1;

    input reset, clock, next, in_noop, in_empty;
    output out_noop, out_empty;

    input [WIDTH-1:0] in_data;
    output [WIDTH-1:0] out_data;

    reg out_noop, out_empty;
    reg [WIDTH-1:0] out_data;

    always @ (posedge clock)
        if (reset) begin
            out_data <= {WIDTH{1'b0}};
            out_noop <= 1'b1;
            out_empty <= 1'b0;
        end else if (next) begin
            out_data <= in_data;
            out_noop <= in_noop;
            out_empty <= in_empty;
        end
endmodule

module reorder_block(in_triangle, out_triangle, out_noop);
    input [`TRIANGLE_BITS-1:0] in_triangle;

    // outgoing triangles
    output [`TRIANGLE_BITS-1:0] out_triangle;
    output out_noop;

    // split up inputs
    wire [`POINT_BITS-1:0] in_a, in_b, in_c;
    wire [`COORD_BITS-1:0] a_x, a_y, a_z, b_x, b_y, b_z, c_x, c_y, c_z;
    wire [`COLOR_BITS-1:0] in_rgb;

    assign {in_a, in_b, in_c, in_rgb} = in_triangle;
    assign {a_x, a_y, a_z} = in_a;
    assign {b_x, b_y, b_z} = in_b;
    assign {c_x, c_y, c_z} = in_c;

    // compare the y coordinates
    wire a_lt_b, a_lt_c, b_lt_c;
    signed_lt #(`COORD_BITS) slt0 (a_y, b_y, a_lt_b);
    signed_lt #(`COORD_BITS) slt1 (a_y, c_y, a_lt_c);
    signed_lt #(`COORD_BITS) slt2 (b_y, c_y, b_lt_c);

    // assign triangle output
    reg [`TRIANGLE_BITS-1:0] out_triangle;
    always @(a_lt_b, a_lt_c, b_lt_c, in_a, in_b, in_c, in_rgb)
        if (!a_lt_b && ! a_lt_c)
            out_triangle = {in_a, in_b, in_c, in_rgb};
```

```
        else if (!b_lt_c)
            out_triangle = {in_b, in_c, in_a, in_rgb};
        else
            out_triangle = {in_c, in_a, in_b, in_rgb};

    // assing noop output
    assign out_noop = 1'b0;
endmodule

module rotation_block(
    in_triangle, in_normal,
    out_triangle, out_normal, out_noop,
    rotation_matrix);

    input [`TRIANGLE_BITS-1:0] in_triangle;
    input [`NORMAL_BITS-1:0] in_normal;

    // outgoing triangles
    output [`TRIANGLE_BITS-1:0] out_triangle;
    output [`NORMAL_BITS-1:0] out_normal;
    output out_noop;

    input [9*`TRIG_OUT_BITS-1:0] rotation_matrix;

    // split up inputs and outputs
    wire [9*`COORD_BITS-1:0] in_m, out_m;
    wire [9*(`COORD_BITS+`TRIG_OUT_BITS+2)-1:0] temp_m;
    wire [3*(`COORD_BITS+`TRIG_OUT_BITS+2)-1:0] temp_normal;
    wire [`COLOR_BITS-1:0] in_rgb, out_rgb;

    assign {in_m, in_rgb} = in_triangle;
    assign out_triangle = {out_m, out_rgb};
    assign out_rgb = in_rgb;

    matrix_product #(`COORD_BITS, `TRIG_OUT_BITS)
        mp (in_m, rotation_matrix, temp_m);
    matrix_trim #(`COORD_BITS+`TRIG_OUT_BITS+2, `COORD_BITS, `TRIG_OUT_BITS-2)
        mt (temp_m, out_m, out_noop);

    vector_matrix_product #(`COORD_BITS, `TRIG_OUT_BITS)
        vmp (in_normal, rotation_matrix, temp_normal);
    vector_trim #(`COORD_BITS+`TRIG_OUT_BITS+2, `COORD_BITS, `TRIG_OUT_BITS-2)
        vt (temp_normal, out_normal);
endmodule

module translation_block(
    in_triangle, out_triangle, out_noop, translation_vector);
    input [`TRIANGLE_BITS-1:0] in_triangle;

    // outgoing triangles
    output [`TRIANGLE_BITS-1:0] out_triangle;
    output out_noop;

    input [3*`COORD_BITS-1:0] translation_vector;

    // split up inputs and outputs
    wire [9*`COORD_BITS-1:0] in_m, out_m;
    wire [9*(`COORD_BITS+1)-1:0] temp_m;
    wire [`COLOR_BITS-1:0] in_rgb, out_rgb;

    assign {in_m, in_rgb} = in_triangle;
    assign out_triangle = {out_m, out_rgb};
    assign out_rgb = in_rgb;
```



```

    matrix_subtract #(`COORD_BITS)
        ms (in_m, {3{translation_vector}}, temp_m);
    matrix_trim #(`COORD_BITS+1, `COORD_BITS)
        mt (temp_m, out_m, out_noop);
endmodule

module zoom_block(in_triangle, out_triangle, out_noop, zoom);
    input [`TRIANGLE_BITS-1:0] in_triangle;
    input [`COORD_BITS-1:0] zoom;

    localparam TEMP_MAX = 2*`COORD_BITS-2;
    localparam TEMP_MIN = `COORD_BITS-2;

    // outgoing triangles
    output [`TRIANGLE_BITS-1:0] out_triangle;
    output out_noop;

    // split up inputs
    wire [`POINT_BITS-1:0] in_a, in_b, in_c;
    wire [`COORD_BITS-1:0] in_a_x, in_a_y, in_a_z,
                           in_b_x, in_b_y, in_b_z,
                           in_c_x, in_c_y, in_c_z;
    wire [`COLOR_BITS-1:0] in_rgb;

    assign {in_a, in_b, in_c, in_rgb} = in_triangle;
    assign {in_a_x, in_a_y, in_a_z} = in_a;
    assign {in_b_x, in_b_y, in_b_z} = in_b;
    assign {in_c_x, in_c_y, in_c_z} = in_c;

    wire [`POINT_BITS-1:0] out_a, out_b, out_c;
    wire [`COORD_BITS-1:0] out_a_x, out_a_y, out_a_z,
                           out_b_x, out_b_y, out_b_z,
                           out_c_x, out_c_y, out_c_z;
    wire [`COLOR_BITS-1:0] out_rgb;

    assign out_a = {out_a_x, out_a_y, out_a_z};
    assign out_b = {out_b_x, out_b_y, out_b_z};
    assign out_c = {out_c_x, out_c_y, out_c_z};
    assign out_triangle = {out_a, out_b, out_c, out_rgb};

    assign out_rgb = in_rgb;
    assign out_noop = 0;

    wire [2*`COORD_BITS-1:0] temp_a_x, temp_a_y, temp_a_z,
                             temp_b_x, temp_b_y, temp_b_z,
                             temp_c_x, temp_c_y, temp_c_z;

    signed_multiply #(`COORD_BITS, `COORD_BITS)
        sm_a_x (in_a_x, zoom, temp_a_x);
    signed_multiply #(`COORD_BITS, `COORD_BITS)
        sm_a_y (in_a_y, zoom, temp_a_y);
    signed_multiply #(`COORD_BITS, `COORD_BITS)
        sm_a_z (in_a_z, zoom, temp_a_z);

    signed_multiply #(`COORD_BITS, `COORD_BITS)
        sm_b_x (in_b_x, zoom, temp_b_x);
    signed_multiply #(`COORD_BITS, `COORD_BITS)
        sm_b_y (in_b_y, zoom, temp_b_y);
    signed_multiply #(`COORD_BITS, `COORD_BITS)
        sm_b_z (in_b_z, zoom, temp_b_z);

    signed_multiply #(`COORD_BITS, `COORD_BITS)
```

triangle_pipeline.v

```
        sm_c_x (in_c_x, zoom, temp_c_x);
    signed_multiply #(`COORD_BITS, `COORD_BITS)
        sm_c_y (in_c_y, zoom, temp_c_y);
    signed_multiply #(`COORD_BITS, `COORD_BITS)
        sm_c_z (in_c_z, zoom, temp_c_z);

    assign out_a_x = temp_a_x[TEMP_MAX:TEMP_MIN];
    assign out_a_y = temp_a_y[TEMP_MAX:TEMP_MIN];
    assign out_a_z = temp_a_z[TEMP_MAX:TEMP_MIN];

    assign out_b_x = temp_b_x[TEMP_MAX:TEMP_MIN];
    assign out_b_y = temp_b_y[TEMP_MAX:TEMP_MIN];
    assign out_b_z = temp_b_z[TEMP_MAX:TEMP_MIN];

    assign out_c_x = temp_c_x[TEMP_MAX:TEMP_MIN];
    assign out_c_y = temp_c_y[TEMP_MAX:TEMP_MIN];
    assign out_c_z = temp_c_z[TEMP_MAX:TEMP_MIN];
endmodule
```


triangle_source.v

```
`timescale 1ns / 1ps
`include "defines.v"
module triangle_source(
        reset, clock, model_select, triangle_data,
        triangle_empty, triangle_noop, next, next_frame);
   parameter ADDRESS_BITS = 12;

   input reset;
   input clock;
   output [`TRIANGLE_BITS + `NORMAL_BITS - 1:0] triangle_data;
   output triangle_empty;
   output triangle_noop;
   input next;
   input next_frame;
   input [3:0] model_select;

   wire [ADDRESS_BITS-1:0] address0, address1;
   wire [`TRIANGLE_BITS + `NORMAL_BITS - 1:0] triangle_data0, triangle_data1;
   wire triangle_empty0, triangle_empty1,
        triangle_noop0, triangle_noop1;

   assign {triangle_data, triangle_empty, triangle_noop} = model_select[0] ?
          {triangle_data0, triangle_empty0, triangle_noop0} :
          {triangle_data1, triangle_empty1, triangle_noop1};

   rom_data_source #(`TRIANGLE_BITS + `NORMAL_BITS, ADDRESS_BITS) rds0 (
        .clock(clock),
        .reset(reset),
        .next_address(address0),
        .data(triangle_data0),
        .empty(triangle_empty0),
        .noop(triangle_noop0),
        .next(next),
        .next_frame(next_frame)
    );

    triangle_data td0 (.addr(address0), .clk(clock), .dout(triangle_data0));

    rom_data_source #(`TRIANGLE_BITS + `NORMAL_BITS, ADDRESS_BITS) rds1 (
        .clock(clock),
        .reset(reset),
        .next_address(address1),
        .data(triangle_data1),
        .empty(triangle_empty1),
        .noop(triangle_noop1),
        .next(next),
        .next_frame(next_frame)
    );

    triangle_data1 td1 (.addr(address1), .clk(clock), .dout(triangle_data1));
endmodule

module rom_data_source(
        reset, clock, next_address, data,
        empty, noop, next, next_frame);
    parameter DATA_BITS = 60;
    parameter ADDRESS_BITS = 10;

    input reset;
    input clock;
    output [ADDRESS_BITS-1:0] next_address;
        // save a cycle by outputting next/address
    input [DATA_BITS-1:0] data;
```

triangle_source.v

```
    output empty;
    output noop;
    input next;
    input next_frame;

    //output [ADDRESS_BITS-1:0] max_address;


    // register output
    reg empty;
    reg noop;
    reg [ADDRESS_BITS-1:0] address;
    reg [ADDRESS_BITS-1:0] max_address;

    // wiring
    wire next_empty;
    wire next_noop;
    wire [ADDRESS_BITS-1:0] next_address;
    wire [ADDRESS_BITS-1:0] next_max_address;


    always @ (posedge clock)
        if (reset) begin
            address <= 0;
            empty <= 1'b0;
            noop <= 1'b1;
            max_address <= -1;
        end else if (next_frame) begin
            address <= 1;
            empty <= 0;
            noop <= 1;
            max_address <= max_address;
        end else begin
            address <= next_address;
            empty <= next_empty;
            noop <= next_noop;
            max_address <= next_max_address;
        end

    assign next_address =
        ((address <= max_address) && next && !reset) ? address + 1: address;
    assign next_empty =
        next_address > max_address;
    assign next_noop =
        next_address == 0 || next_empty;
    assign next_max_address =
        (address == 0) ? data[ADDRESS_BITS-1:0] : max_address;
endmodule
```


orienter.v

```
`timescale 1ns / 1ps
`include "defines.v"
module orienter(reset, clock,
                mouse0_clock, mouse0_data,
                mouse1_clock, mouse1_data, buttons,
                enable, switch_buffer,
                paddle_y, ball_x, ball_y,
                translation_vector,
                rotation_matrix,
                light_vector,
                zoom,
                rotor_matrix,
                ball_vector,
                paddle_vector);

    input reset;
    input clock;

    input mouse0_clock;
    input mouse0_data;
    input mouse1_clock;
    input mouse1_data;
    input [`NUM_BUTTONS-1:0] buttons;

    input enable;
    input switch_buffer;

    input [9:0] paddle_y, ball_x, ball_y;

    localparam MAX = {2'b01, {(`COORD_BITS-2){1'b0}}};
    localparam ZERO = {(`COORD_BITS){1'b0}};

    output [3*`COORD_BITS-1:0] translation_vector, ball_vector,
                               paddle_vector, light_vector;
    output [9*`TRIG_OUT_BITS-1:0] rotation_matrix, rotor_matrix;
    output [`COORD_BITS-1:0] zoom;

    // calculate ball/paddle positions
    wire [10:0] neg_paddle_y, neg_ball_x, neg_ball_y;
    signed_subtract #(11) snpy (8, {1'b0, paddle_y}, neg_paddle_y);
    signed_subtract #(11) snbx ((28+8+16), {1'b0, ball_x}, neg_ball_x);
    signed_subtract #(11) snby (8, {1'b0, ball_y}, neg_ball_y);

    wire [3*`COORD_BITS-1:0] next_ball_vector, next_paddle_vector;
    assign next_paddle_vector =
        {ZERO, {neg_paddle_y[10:0], 1'b0}, ZERO};
    assign next_ball_vector =
        {{neg_ball_x[10:0], 1'b0}, {neg_ball_y[10:0], 1'b0}, ZERO};

    assign light_vector = {ZERO, {2'b11, {(`COORD_BITS-2){1'b0}}}, ZERO};
    // maintain zoom
    wire [`COORD_BITS-1:0] temp_zoom;
    reg [`COORD_BITS-1:0] zoom;
    slider #(`COORD_BITS, 1, MAX, MAX/2, 1) sl_zoom
        (reset, clock, enable,
        buttons[6]&&buttons[8], buttons[7]&&buttons[8], temp_zoom);

    // maintain transpose
    wire [`COORD_BITS-1:0] t_x, t_y, t_z;
    wire [3*`COORD_BITS-1:0] temp_translation_vector;
    reg [3*`COORD_BITS-1:0] translation_vector, paddle_vector, ball_vector;
    assign temp_translation_vector = {t_x, t_y, t_z};
    assign t_z = 0;
```

```

    slider #(`COORD_BITS, -13'd1000, 13'd1000, 0, 12'b1) slx
        (reset, clock, enable, buttons[1]&&buttons[8],
        buttons[0]&&buttons[8], t_x);
    slider #(`COORD_BITS, -13'd1000, 13'd1000, 0, 12'b1) sly
        (reset, clock, enable, buttons[2]&&buttons[8],
        buttons[3]&&buttons[8], t_y);


    // maintain rotation
    wire [`TRIG_IN_BITS-1:0] theta_x, theta_y, theta_z, theta_rotor;
    slider #(`TRIG_IN_BITS, -128, 127, 0, 1, 1) sl_theta_x
        (reset, clock, enable, buttons[1]&&(!buttons[8]),
        buttons[0]&&(!buttons[8]), theta_x);
    slider #(`TRIG_IN_BITS, -128, 127, -128, 1, 1) sl_theta_y
        (reset, clock, enable, buttons[2]&&(!buttons[8]),
        buttons[3]&&(!buttons[8]), theta_y);
    slider #(`TRIG_IN_BITS, -128, 127, -128, 1, 1) sl_theta_z
        (reset, clock, enable, buttons[6]&&(!buttons[8]),
        buttons[7]&&(!buttons[8]), theta_z);
    slider #(`TRIG_IN_BITS, -128, 127, 0, 3, 1) sl_theta_rotor
        (reset, clock, enable, 1'b0, 1'b1, theta_rotor);

    wire [9*`TRIG_OUT_BITS-1:0] temp_rotation_matrix, temp_rotor_matrix;
    reg [9*`TRIG_OUT_BITS-1:0] rotation_matrix, rotor_matrix;
    thetas_to_matrix ttm
        (reset, clock, enable, theta_x, theta_y, theta_z,
         theta_rotor, temp_rotation_matrix, temp_rotor_matrix);

    always @ (posedge clock)
        if (reset) begin
            rotation_matrix <= {(9*`TRIG_OUT_BITS){1'b0}};
            rotor_matrix <= {(9*`TRIG_OUT_BITS){1'b0}};
            translation_vector <= {(3*`COORD_BITS){1'b0}};
            paddle_vector <= {(3*`COORD_BITS){1'b0}};
            ball_vector <= {(3*`COORD_BITS){1'b0}};
            zoom <= MAX/2;
        end else if (switch_buffer) begin
            zoom <= temp_zoom;
            rotation_matrix <= temp_rotation_matrix;
            rotor_matrix <= temp_rotor_matrix;
            translation_vector <= temp_translation_vector;
            paddle_vector <= next_paddle_vector;
            ball_vector <= next_ball_vector;
        end
endmodule

module slider(reset, clock, enable, up, down, value);
    parameter WIDTH = 12;
    parameter MIN_VALUE = -100;
    parameter MAX_VALUE = 100;
    parameter START_VALUE = 0;
    parameter DELTA = 1;
    parameter WRAP = 0;

    input reset;
    input clock;
    input enable;
    input up;
    input down;
    output [WIDTH-1:0] value;

    // registered state
```



```
    reg [WIDTH-1:0] value;

    // next holders
    reg [WIDTH-1:0] next_value;

    wire [WIDTH:0] sum, diff;
    wire max_lt_sum, diff_lt_min;


    always @ (posedge clock)
            if (reset)
                value <= START_VALUE;
            else
                value <= next_value;

    always @(up or down or value or sum or diff or
             enable or max_lt_sum or diff_lt_min)
        begin
            // by default, don't change state
            next_value = value;
            if (enable && (up ^ down)) begin
                if (up)
                    if (!max_lt_sum)
                        next_value = sum[WIDTH-1:0];
                    else
                        next_value = WRAP ? MIN_VALUE : MAX_VALUE;
                else
                    if (!diff_lt_min)
                        next_value = diff[WIDTH-1:0];
                    else
                        next_value = WRAP ? MAX_VALUE : MIN_VALUE;
            end
        end

    signed_add #(WIDTH)      sa (value, DELTA, sum);
    signed_subtract #(WIDTH) ss (value, DELTA, diff);

    signed_lt #(WIDTH+1) slt0 (MAX_VALUE, sum, max_lt_sum);
    signed_lt #(WIDTH+1) slt1 (diff, MIN_VALUE, diff_lt_min);
endmodule

```

game_fsm.v

```
`timescale 1ns / 1ps
module fsm (reset, clock, enable, speed, paddle_speed,
            up_sync, down_sync, paddle_y, ball_x, ball_y);
    input reset;
    input clock;
    input enable;
    input [3:0] speed;
    input [3:0] paddle_speed;
    input up_sync;
    input down_sync;
    output [9:0] paddle_y;
    output [9:0] ball_x;
    output [9:0] ball_y;

    // registered state
    reg [0:0] state;
    reg [6:0] speed_x;
    reg [6:0] speed_y;
    reg dir_x;
    reg dir_y;

    // Parameters
    `include "params.v"
    parameter SPEED_GRADIANT = 3;
    parameter MAX_DIST = (PADDLE_HEIGHT + BALL_HEIGHT) / 2;

    //  States
    parameter PLAYING = 0;
    parameter GAME_OVER = 1;

    // next state holders
    reg [0:0] next_state;
    reg [6:0] next_speed_x;
    reg [6:0] next_speed_y;
    reg next_dir_x;
    reg next_dir_y;

    // Minor FSMs (Used mainly for abstraction)
    paddle_fsm paddle_fsm1
        (clock, reset, enable, up_sync, down_sync, paddle_speed, paddle_y);
    ball_fsm  ball_fsm1
        (reset, clock, enable, next_speed_x, next_speed_y,
        next_dir_x, next_dir_y, ball_x, ball_y);

    // state changes synchronously
    always @ (posedge clock)
        if (reset) begin
            state <= PLAYING;
            speed_x <= {4'b0, speed[1:0]};
            speed_y <= {4'b0, speed[3:2]};
            dir_x <= POS;
            dir_y <= POS;
        end else begin
            state <= next_state;
            speed_x <= next_speed_x;
            speed_y <= next_speed_y;
            dir_x <= next_dir_x;
            dir_y <= next_dir_y;
        end

    // local wires used for calculation
    wire [9:0] ball_center_y = ball_y + (BALL_HEIGHT / 2);
    wire [9:0] paddle_center_y = paddle_y + (PADDLE_HEIGHT / 2);
```

Page: 1

game_fsm.v

```
    wire [9:0] dist = ball_center_y < paddle_center_y ?
                            paddle_center_y - ball_center_y :
                            ball_center_y - paddle_center_y;
    wire [3:0] y_speed_change =
        (SPEED_GRADIANT * dist) / (PADDLE_HEIGHT/2)/*MAX_DIST*/;
    wire [3:0] x_speed_change = SPEED_GRADIANT - y_speed_change;
    wire y_speed_change_dir =
        ball_center_y < paddle_center_y ? NEG : POS;

    // next state calculated asynchronously
    always @ (enable or state or ball_x or ball_y or
              speed_x or speed_y or dir_x or dir_y
              or dist or x_speed_change or y_speed_change
              or y_speed_change_dir) begin
        // by default, keep old state
        next_state = state;
        next_speed_x = speed_x;
        next_speed_y = speed_y;
        next_dir_x = dir_x;
        next_dir_y = dir_y;

        case (state)
            PLAYING:
                if (enable) begin
                    if (dir_x == POS &&
                        ball_x + speed_x >=
                        DISPLAY_WIDTH - BORDER_WIDTH - BALL_WIDTH)
                        next_dir_x = NEG; // Right Wall

                    if (dir_y == POS &&
                        ball_y + speed_y >=
                        DISPLAY_HEIGHT - BORDER_HEIGHT - BALL_HEIGHT)
                        next_dir_y = NEG; // Buttom Wall

                    if (dir_y == NEG && ball_y <= BORDER_HEIGHT + speed_y)
                        next_dir_y = POS; // Top Wall

                    // see if we hit the paddle
                    // (ignoring speed_y, but compensating for speed_x)
                    if (dir_x == NEG
                        && ball_x <= PADDLE_X + PADDLE_WIDTH + speed_x
                        && ball_x + BALL_WIDTH + speed_x >= PADDLE_X
                        && dist < MAX_DIST) begin

                        next_dir_x = POS;
                        next_speed_x = speed_x + x_speed_change;

                        // since we're using unsigned arithmetic,
                        // speed_y is tricky
                        if (dir_y == y_speed_change_dir)
                            next_speed_y = speed_y + y_speed_change;
                        else if (speed_y < y_speed_change) begin
                            next_speed_y = y_speed_change - speed_y;
                            next_dir_y = ~next_dir_y;
                        end else
                            next_speed_y = speed_y - y_speed_change;

                        // Make sure we don't get too close to overflow
                        if (next_speed_x > MAX_SPEED)
                            next_speed_x = MAX_SPEED;
                        if (next_speed_y > MAX_SPEED)
                            next_speed_y = MAX_SPEED;
                    end
```

Page: 2

```
                    if (dir_x == NEG && ball_x <= speed_x) begin
                        // Hit left screen
                        next_state = GAME_OVER;
                        next_speed_x = 0;
                        next_speed_y = 0;
                    end
                end
            GAME_OVER:
                if (up_sync && down_sync) begin
                    next_state = PLAYING;
                    next_speed_x = {4'b0, speed[1:0]};
                    next_speed_y = {4'b0, speed[3:2]};
                    next_dir_x = POS;
                    next_dir_y = POS;
                end
        endcase
    end
endmodule

module ball_fsm(reset, clock, enable, speed_x,
                speed_y, dir_x, dir_y, ball_x, ball_y);
    input reset;
    input clock;
    input enable;
    input [6:0] speed_x;
    input [6:0] speed_y;
    input dir_x;
    input dir_y;
    output [9:0] ball_x;
    output [9:0] ball_y;

    // Parameters
    `include "params.v"

    // State
    reg [9:0] ball_x;
    reg [9:0] ball_y;

    // Next State
    wire [9:0] next_x;
    wire [9:0] next_y;

    always @ (posedge clock)
        if (reset) begin
            ball_x <= PADDLE_X + PADDLE_WIDTH;
            ball_y <= DISPLAY_HEIGHT / 2;
        end else begin
            ball_x <= next_x;
            ball_y <= next_y;
        end

    assign next_x = enable ?
        (dir_x == POS ? ball_x + speed_x : ball_x - speed_x) :
        ball_x;
    assign next_y = enable ?
        (dir_y == POS ? ball_y + speed_y : ball_y - speed_y) :
        ball_y;
endmodule

module paddle_fsm(clock, reset, enable, up_sync,
                  down_sync, paddle_speed, paddle_y);
    input clock;
```



```
    input reset;
    input enable;
    input up_sync;
    input down_sync;
    input [3:0] paddle_speed;
    output [9:0] paddle_y;

    // parameters
    `include "params.v"
    parameter MIN_Y = BORDER_HEIGHT;
    parameter MAX_Y = DISPLAY_HEIGHT - BORDER_HEIGHT - PADDLE_HEIGHT;
    parameter START_Y = (MIN_Y + MAX_Y)/2;

    // registered state
    reg [9:0] paddle_y;

    // next holders
    reg [9:0] next_y;

    always @ (posedge clock)
            if (reset)
                paddle_y <= START_Y;
            else
                paddle_y <= next_y;

    always @(up_sync or down_sync or paddle_y or enable or paddle_speed)
        begin
            // by default, don't change state
            next_y = paddle_y;
            if (enable && (up_sync ^ down_sync)) begin
                if (up_sync)
                    if (paddle_y > MIN_Y + paddle_speed)
                        next_y = paddle_y - paddle_speed;
                    else
                        next_y = MIN_Y;
                else
                    if (paddle_y + paddle_speed < MAX_Y)
                        next_y = paddle_y + paddle_speed;
                    else
                        next_y = MAX_Y;
            end
        end
endmodule
```

```
`timescale 1ns / 1ps
module display_field(reset, pixel_count, line_count,
                     paddle_y, ball_x, ball_y, in_rgb, out_rgb);
    input reset; // ignored, since display_field has no state
    input [9:0] pixel_count;
    input [9:0] line_count;
    input [9:0] paddle_y;
    input [9:0] ball_x;
    input [9:0] ball_y;
    input [23:0] in_rgb;
    output [23:0] out_rgb;

    // Internal Wiring
    wire [23:0] rgb0, rgb1, rgb2, rgb3, rgb4, rgb5, rgb6,
                rgb7, rgb8, rgb9, rgb10, rgb11, rgb12;

    // Global Parameters
    `include "params.v"

    // Borders
    parameter BACKGROUND    = 0;
    parameter FOREGROUND    = -1;
    assign rgb0 = in_rgb;//BACKGROUND;

    box border1(pixel_count, line_count, 10'b0, 10'b0, rgb0, rgb1);
    defparam border1.RGB = FOREGROUND  ;
    defparam border1.WIDTH  = DISPLAY_WIDTH  ;
    defparam border1.HEIGHT   = BORDER_HEIGHT  ;

    box border2(pixel_count, line_count,
        DISPLAY_WIDTH   - BORDER_WIDTH , 10'b0, rgb1, rgb2);
    defparam border2.RGB = FOREGROUND  ;
    defparam border2.WIDTH  = BORDER_WIDTH ;
    defparam border2.HEIGHT   = DISPLAY_HEIGHT   ;

    box border3(pixel_count, line_count, 10'b0,
        DISPLAY_HEIGHT    - BORDER_HEIGHT  , rgb2, rgb3);
    defparam border3.RGB = FOREGROUND  ;
    defparam border3.WIDTH  = DISPLAY_WIDTH  ;
    defparam border3.HEIGHT   = BORDER_HEIGHT  ;

    // Logo
    parameter MIT_RED = {8'b0101_1111, 8'b0001_1111, 8'b0001_1111};
    parameter MIT_GRAY = {8'b0100_1111, 8'b0100_1111, 8'b0011_1111};
    parameter LOGO_WIDTH  = 10'd39;
    parameter LOGO_HEIGHT   = 10'd195;
    parameter LOGO_X = 10'd133;
    parameter LOGO_Y = 10'd134;
    parameter LOGO_DX = LOGO_WIDTH  + 10'd25;

    box m1(pixel_count, line_count, LOGO_X, LOGO_Y, rgb3, rgb4);
    defparam m1.RGB = MIT_RED;
    defparam m1.WIDTH  = LOGO_WIDTH ;
    defparam m1.HEIGHT   = LOGO_HEIGHT  ;

    box m2(pixel_count, line_count, LOGO_X + LOGO_DX, LOGO_Y, rgb4, rgb5);
    defparam m2.RGB = MIT_RED;
    defparam m2.WIDTH  = LOGO_WIDTH ;
    defparam m2.HEIGHT   = 137;

    box m3(pixel_count, line_count,
            (LOGO_X + 2'd2*LOGO_DX), LOGO_Y, rgb5, rgb6);
    defparam m3.RGB = MIT_RED;
```



```
    defparam m3.WIDTH  = LOGO_WIDTH ;
    defparam m3.HEIGHT   = LOGO_HEIGHT  ;

    box i1(pixel_count, line_count,
            (LOGO_X + 2'd3*LOGO_DX), LOGO_Y, rgb6, rgb7);
    defparam i1.RGB = MIT_RED;
    defparam i1.WIDTH  = LOGO_WIDTH ;
    defparam i1.HEIGHT   = LOGO_WIDTH ;

    box i2(pixel_count, line_count,
            (LOGO_X + 2'd3*LOGO_DX),
            LOGO_Y + 2'd2*LOGO_WIDTH , rgb7, rgb8);
    defparam i2.RGB = MIT_GRAY;
    defparam i2.WIDTH  = LOGO_WIDTH ;
    defparam i2.HEIGHT   = LOGO_HEIGHT   - 2*LOGO_WIDTH ;

    box t1(pixel_count, line_count,
            (LOGO_X + 3'd4*LOGO_DX), LOGO_Y, rgb8, rgb9);
    defparam t1.RGB = MIT_RED;
    defparam t1.WIDTH  = LOGO_HEIGHT   - 2*LOGO_WIDTH ;
    defparam t1.HEIGHT   = LOGO_WIDTH ;

    box t2(pixel_count, line_count,
            (LOGO_X + 3'd4*LOGO_DX),
            (LOGO_Y + 2'd2*LOGO_WIDTH ), rgb9, rgb10);
    defparam t2.RGB = MIT_RED;
    defparam t2.WIDTH  = LOGO_WIDTH ;
    defparam t2.HEIGHT   = LOGO_HEIGHT   - 2*LOGO_WIDTH ;

    // paddle
    box paddle(pixel_count, line_count,
               PADDLE_X, paddle_y, rgb10, rgb11);
    defparam paddle.RGB = FOREGROUND  ;
    defparam paddle.WIDTH  = PADDLE_WIDTH ;
    defparam paddle.HEIGHT   = PADDLE_HEIGHT  ;

    // ball
    box ball(pixel_count, line_count, ball_x, ball_y, rgb11, rgb12);
    defparam ball.RGB = FOREGROUND  ;
    defparam ball.WIDTH  = BALL_WIDTH ;
    defparam ball.HEIGHT   = BALL_HEIGHT  ;

    assign out_rgb = rgb12;
endmodule

module box(pixel_count, line_count, x, y, rgb_in, rgb_out);
    input [9:0] pixel_count;
    input [9:0] line_count;
    input [9:0] x;
    input [9:0] y;
    input [23:0] rgb_in;
    output [23:0] rgb_out;

    parameter WIDTH  = 1;
    parameter HEIGHT   = 1;
    parameter RGB = 24'b0;

    assign rgb_out =
        ((pixel_count >= x && pixel_count < x + WIDTH )
        && (line_count >= y && line_count < y + HEIGHT  ))
        ? RGB : rgb_in;
endmodule
```

```
module signed_add(a, b, c);
    parameter WIDTH = 12;

    input signed [WIDTH-1:0] a, b;
    output signed [WIDTH:0] c;

    assign c = a + b;
endmodule

module signed_add3(a, b, c, sum);
    parameter WIDTH = 12;
    localparam C_WIDTH = WIDTH + 2;

    input signed [WIDTH-1:0] a, b, c;
    output signed [C_WIDTH-1:0] sum;

    assign sum = a + b + c;
endmodule

module signed_inc(a, c);
    parameter WIDTH = 12;

    input signed [WIDTH-1:0] a;
    output signed [WIDTH:0] c;

    assign c = a + 1;
endmodule

module signed_lt(a, b, c);
    parameter WIDTH = 12;

    input signed [WIDTH-1:0] a, b;
    output c;

    assign c = a < b;
endmodule

module signed_bound(a, b);
    parameter WIDTH = 12;
    parameter MAX = 100;
    parameter MIN = -100;

    input signed [WIDTH-1:0] a;
    output signed [WIDTH-1:0]  b;
    assign b = a < MIN ? MIN : (a > MAX ? MAX : a);
endmodule

module signed_multiply(a, b, c);
    parameter A_WIDTH = 12;
    parameter B_WIDTH = 12;
    localparam C_WIDTH = A_WIDTH + B_WIDTH;

    input signed [A_WIDTH-1:0] a;
    input signed [B_WIDTH-1:0] b;
    output signed [C_WIDTH-1:0] c;

    assign c = a * b;
endmodule

module signed_negate(a, b);
    parameter WIDTH = 12;

    input signed [WIDTH-1:0] a;
```



```
    output signed [WIDTH:0] b;

    assign b = -a;
endmodule

module signed_outer3(a, b, c, min, max);
    parameter WIDTH = 12;

    input signed [WIDTH-1:0] a, b, c;
    output signed [WIDTH-1:0] min, max;

    assign min = a < b ?
                 (a < c ? a : c):
                 (b < c ? b : c);

    assign max = a > b ?
                 (a > c ? a : c):
                 (b > c ? b : c);
endmodule

module signed_subtract(a, b, c);
    parameter WIDTH = 12;

    input signed [WIDTH-1:0] a, b;
    output signed [WIDTH:0] c;

    assign c = a - b;
endmodule

module signed_to_unsigned(in_a, next_a);
    parameter WIDTH = 10;
    parameter ZERO = 320;
    input [WIDTH-1:0] in_a;
    output [WIDTH-1:0] next_a;

    wire [WIDTH:0] temp;

    signed_add #(WIDTH) sa1 (in_a, ZERO, temp);
    assign next_a = temp[WIDTH-1:0];
endmodule

module signed_trim(in_a, next_a, next_noop);
    parameter IN_WIDTH = 13;
    parameter OUT_WIDTH = 12;
    parameter SKIP_WIDTH = 0;

    input [IN_WIDTH-1:0] in_a;
    output [OUT_WIDTH-1:0] next_a;
    output next_noop;

    wire [OUT_WIDTH-1:0] next_a;
    wire [IN_WIDTH-1:OUT_WIDTH-1] trimmed;
    wire next_noop;

    assign next_a = in_a[OUT_WIDTH-1+SKIP_WIDTH:0+SKIP_WIDTH];
    assign trimmed = in_a[IN_WIDTH-1:OUT_WIDTH-1+SKIP_WIDTH];
    assign next_noop =
        !(trimmed == {(IN_WIDTH-OUT_WIDTH-SKIP_WIDTH){1'b0}}
                  || {(IN_WIDTH-OUT_WIDTH-SKIP_WIDTH){1'b1}});
endmodule
```

vector.v

```
module vector_subtract(a, b, c);
    parameter WIDTH = 12;

    input [3*WIDTH-1:0] a, b;
    output [3*(WIDTH+1)-1:0] c;

    wire [WIDTH-1:0] a_x, a_y, a_z;
    wire [WIDTH-1:0] b_x, b_y, b_z;
    wire [WIDTH:0] c_x, c_y, c_z;

    assign {a_x, a_y, a_z} = a;
    assign {b_x, b_y, b_z} = b;
    assign c = {c_x, c_y, c_z};

    signed_subtract #(WIDTH) ss_x (a_x, b_x, c_x);
    signed_subtract #(WIDTH) ss_y (a_y, b_y, c_y);
    signed_subtract #(WIDTH) ss_z (a_z, b_z, c_z);
endmodule

module vector_matrix_product(a, b, c);
    parameter A_WIDTH = 12;
    parameter B_WIDTH = 12;
    localparam C_WIDTH = A_WIDTH + B_WIDTH + 2;

    input [3*A_WIDTH-1:0] a; /vector
    input [9*B_WIDTH-1:0] b; /matrix
    output [3*C_WIDTH-1:0] c;

    wire [3*B_WIDTH-1:0] b1, b2, b3;
    wire [B_WIDTH-1:0] b1_x, b1_y, b1_z,
                       b2_x, b2_y, b2_z,
                       b3_x, b3_y, b3_z;
    wire [C_WIDTH-1:0] c_x, c_y, c_z;

    assign {b1, b2, b3} = b;
    assign {b1_x, b1_y, b1_z} = b1;
    assign {b2_x, b2_y, b2_z} = b2;
    assign {b3_x, b3_y, b3_z} = b3;

    assign c = {c_x, c_y, c_z};

    dot_product #(A_WIDTH, B_WIDTH) dp1 (a, {b1_x, b2_x, b3_x}, c_x);
    dot_product #(A_WIDTH, B_WIDTH) dp2 (a, {b1_y, b2_y, b3_y}, c_y);
    dot_product #(A_WIDTH, B_WIDTH) dp3 (a, {b1_z, b2_z, b3_z}, c_z);
endmodule

module vector_trim(in_v, next_v, next_noop);
    parameter IN_WIDTH = 12;
    parameter OUT_WIDTH = 11;
    parameter SKIP_WIDTH = 0;

    input [3*IN_WIDTH-1:0] in_v;
    output [3*OUT_WIDTH-1:0] next_v;
    output next_noop;

    wire [IN_WIDTH-1:0] in_x, in_y, in_z;
    wire [OUT_WIDTH-1:0] next_x, next_y, next_z;
    wire x_noop, y_noop, z_noop ;

    assign {in_x, in_y, in_z} = in_v;
    assign next_v = {next_x, next_y, next_z};
    assign next_noop = x_noop||y_noop||z_noop ;

```

Page: 1

vector.v

```
    signed_trim #(IN_WIDTH, OUT_WIDTH, SKIP_WIDTH) st_x (in_x, next_x, x_noop);
    signed_trim #(IN_WIDTH, OUT_WIDTH, SKIP_WIDTH) st_y (in_y, next_y, y_noop);
    signed_trim #(IN_WIDTH, OUT_WIDTH, SKIP_WIDTH) st_z (in_z, next_z, z_noop );
endmodule

module dot_product(a, b, c);
    parameter A_WIDTH = 12;
    parameter B_WIDTH = 12;
    localparam C_WIDTH = A_WIDTH + B_WIDTH + 2;

    input [3*A_WIDTH-1:0] a;
    input [3*B_WIDTH-1:0] b;
    output [C_WIDTH-1:0] c;

    wire [A_WIDTH-1:0] a_x, a_y, a_z;
    wire [B_WIDTH-1:0] b_x, b_y, b_z;
    wire [(A_WIDTH+B_WIDTH)-1:0] c_x, c_y, c_z;

    assign {a_x, a_y, a_z} = a;
    assign {b_x, b_y, b_z} = b;

    signed_multiply #(A_WIDTH, B_WIDTH) sm_x (a_x, b_x, c_x);
    signed_multiply #(A_WIDTH, B_WIDTH) sm_y (a_y, b_y, c_y);
    signed_multiply #(A_WIDTH, B_WIDTH) sm_z (a_z, b_z, c_z);

    signed_add3 #(A_WIDTH+B_WIDTH) sa(c_x, c_y, c_z, c);
endmodule

module cross_product(a, b, c);
    parameter A_WIDTH = 12;
    parameter B_WIDTH = 12;

    input [3*A_WIDTH-1:0] a;
    input [3*B_WIDTH-1:0] b;
    output [3*(A_WIDTH+B_WIDTH+1)-1:0] c;

    wire [A_WIDTH-1:0] a_x, a_y, a_z;
    wire [B_WIDTH-1:0] b_x, b_y, b_z;
    wire [A_WIDTH+B_WIDTH-1:0] c_x1, c_y1, c_z1 , c_x2, c_y2, c_z2 ;

    assign {a_x, a_y, a_z} = a;
    assign {b_x, b_y, b_z} = b;

    signed_multiply #(A_WIDTH, B_WIDTH) sm_x1 (a_y, b_z, c_x1);
    signed_multiply #(A_WIDTH, B_WIDTH) sm_x2 (a_z, b_y, c_x2);

    signed_multiply #(A_WIDTH, B_WIDTH) sm_y1 (a_z, b_x, c_y1);
    signed_multiply #(A_WIDTH, B_WIDTH) sm_y2 (a_x, b_z, c_y2);

    signed_multiply #(A_WIDTH, B_WIDTH) sm_z1  (a_x, b_y, c_z1 );
    signed_multiply #(A_WIDTH, B_WIDTH) sm_z2  (a_y, b_x, c_z2 );

    vector_subtract #(A_WIDTH + B_WIDTH) vs ({c_x1, c_y1, c_z1 }, {c_x2, c_y2, c_z2 }, c);
endmodule

```

Page: 2

matrix.v

```
module matrix_product(a, b, c);
    parameter A_WIDTH = 12;
    parameter B_WIDTH = 12;
    localparam C_WIDTH = A_WIDTH+B_WIDTH+2;

    input [9*A_WIDTH-1:0] a; // matrix
    input [9*B_WIDTH-1:0] b; // matrix
    output [9*C_WIDTH-1:0] c; // matrix

    wire [3*B_WIDTH-1:0] b1, b2, b3;
    wire [B_WIDTH-1:0] b1_x, b1_y, b1_z,
                       b2_x, b2_y, b2_z,
                       b3_x, b3_y, b3_z;

    assign {b1, b2, b3} = b;
    assign {b1_x, b1_y, b1_z} = b1;
    assign {b2_x, b2_y, b2_z} = b2;
    assign {b3_x, b3_y, b3_z} = b3;

    wire [3*C_WIDTH-1:0] c1, c2, c3;
    wire [C_WIDTH-1:0] c1_x, c1_y, c1_z,
                       c2_x, c2_y, c2_z,
                       c3_x, c3_y, c3_z;

    assign c = {c1, c2, c3};
    assign c1 = {c1_x, c1_y, c1_z};
    assign c2 = {c2_x, c2_y, c2_z};
    assign c3 = {c3_x, c3_y, c3_z};

    matrix_vector_product #(A_WIDTH, B_WIDTH) mvpx
        (a, {b1_x, b2_x, b3_x}, {c1_x, c2_x, c3_x});
    matrix_vector_product #(A_WIDTH, B_WIDTH) mvpy
        (a, {b1_y, b2_y, b3_y}, {c1_y, c2_y, c3_y});
    matrix_vector_product #(A_WIDTH, B_WIDTH) mvpz
        (a, {b1_z, b2_z, b3_z}, {c1_z, c2_z, c3_z});
endmodule

module matrix_subtract(a, b, c);
    parameter WIDTH = 12;

    input [9*WIDTH-1:0] a, b;
    output [9*(WIDTH+1)-1:0] c;

    wire [3*WIDTH-1:0] a1, a2, a3, b1, b2, b3;
    wire [3*(WIDTH+1)-1:0] c1, c2, c3;

    assign {a1, a2, a3} = a;
    assign {b1, b2, b3} = b;
    assign c = {c1, c2, c3};

    vector_subtract #(WIDTH) vs_1 (a1, b1, c1);
    vector_subtract #(WIDTH) vs_2 (a2, b2, c2);
    vector_subtract #(WIDTH) vs_3 (a3, b3, c3);
endmodule

module matrix_trim(in_m, next_m, next_noop);
    parameter IN_WIDTH = 12;
    parameter OUT_WIDTH = 11;
    parameter SKIP_WIDTH = 0;

    input [9*IN_WIDTH-1:0] in_m;
    output [9*OUT_WIDTH-1:0] next_m;
    output next_noop;
```

Page: 1

matrix.v

```

    wire [3*IN_WIDTH-1:0] in_a, in_b, in_c;
    wire [3*OUT_WIDTH-1:0] next_a, next_b, next_c;
    wire a_noop, b_noop, c_noop;

    assign {in_a, in_b, in_c} = in_m;
    assign next_m = {next_a, next_b, next_c};
    assign next_noop = a_noop||b_noop||c_noop;

    vector_trim #(IN_WIDTH, OUT_WIDTH, SKIP_WIDTH) st_a (in_a, next_a, a_noop);
    vector_trim #(IN_WIDTH, OUT_WIDTH, SKIP_WIDTH) st_b (in_b, next_b, b_noop);
    vector_trim #(IN_WIDTH, OUT_WIDTH, SKIP_WIDTH) st_c (in_c, next_c, c_noop);
endmodule

module matrix_vector_product(a, b, c);
    parameter A_WIDTH = 12;
    parameter B_WIDTH = 12;
    localparam C_WIDTH = A_WIDTH + B_WIDTH + 2;

    input [9*A_WIDTH-1:0] a; // matrix
    input [3*B_WIDTH-1:0] b; // vector
    output [3*C_WIDTH-1:0] c;

    wire [3*A_WIDTH-1:0] a1, a2, a3;
    wire [C_WIDTH-1:0] c_x, c_y, c_z;

    assign {a1, a2, a3} = a;
    assign c = {c_x, c_y, c_z};

    dot_product #(A_WIDTH, B_WIDTH) dp1 (a1, b, c_x);
    dot_product #(A_WIDTH, B_WIDTH) dp2 (a2, b, c_y);
    dot_product #(A_WIDTH, B_WIDTH) dp3 (a3, b, c_z);
endmodule
```

Page: 2

thetas_to_matrix.v

```
`timescale 1ns / 1ps
`include "defines.v"
module thetas_to_matrix(
        reset, clock, enable, theta_x, theta_y, theta_z,
        theta_rotor, matrix, rotor_matrix);
    input reset, clock, enable;

    input [`TRIG_IN_BITS-1:0] theta_x, theta_y, theta_z, theta_rotor;
    output [9*`TRIG_OUT_BITS-1:0] matrix, rotor_matrix;

    localparam IDLE = 0;
    localparam X_ROTATION = 1;
    localparam Y_ROTATION  = 2;
    localparam Z_ROTATION = 3;
    localparam ROTOR = 4;
    localparam FINAL  = 5;

    reg [2:0] state, next_state;
    reg [`TRIG_IN_BITS-1:0] theta, next_theta;
    reg [1:0] select, next_select;
    reg [9*`TRIG_OUT_BITS-1:0] matrix, next_matrix,
                               rotor_matrix, next_rotor_matrix,
                               temp0, next_temp0,
                               temp1, next_temp1;

    wire [9*`TRIG_OUT_BITS-1:0] ttm_matrix, mt_matrix;
    wire [9*(2*`TRIG_OUT_BITS+2)-1:0] mp_matrix;

    theta_to_matrix ttm (theta, select, ttm_matrix);

    matrix_product #(`TRIG_OUT_BITS, `TRIG_OUT_BITS)
        mp (temp0, temp1, mp_matrix);
    matrix_trim
        #(`TRIG_OUT_BITS+`TRIG_OUT_BITS+2, `TRIG_OUT_BITS, `TRIG_OUT_BITS-2)
        mt (mp_matrix, mt_matrix, out_noop);

    always @ (posedge clock)
        if (reset) begin
            state <= IDLE;
            theta <=  {(`TRIG_IN_BITS){1'b0}};
            select <= 2'b0;
            temp0 <= {(9*`TRIG_OUT_BITS){1'b0}};
            temp1 <= {(9*`TRIG_OUT_BITS){1'b0}};
            matrix <= {(9*`TRIG_OUT_BITS){1'b0}};
            rotor_matrix <= {(9*`TRIG_OUT_BITS){1'b0}};
        end else begin
            state <= next_state;
            theta <= next_theta;
            select <= next_select;
            temp0 <= next_temp0;
            temp1 <= next_temp1;
            matrix <= next_matrix;
            rotor_matrix <= next_rotor_matrix;
        end

    wire timer;
    divider #(5) div0 (reset, clock, timer);

    always @ (state, theta, select, temp0, temp1, matrix, enable,
              theta_x, theta_y, ttm_matrix, theta_z, mt_matrix,
              rotor_matrix, theta_rotor)  begin
        next_state = state;
        next_theta = theta;
```


thetas_to_matrix.v

```
        next_select = select;
        next_temp0 = temp0;
        next_temp1 = temp1;
        next_matrix = matrix;
        next_rotor_matrix = rotor_matrix;
        if (timer || enable) begin
            case (state)
                IDLE:
                if (enable) begin
                    next_state = X_ROTATION;
                    next_theta = theta_x;
                    next_select = 2'b0;
                end
                X_ROTATION:
                begin
                    next_state = Y_ROTATION ;
                    next_theta = theta_y;
                    next_select = 2'b1;
                    next_temp0 = ttm_matrix;
                end
                Y_ROTATION :
                begin
                    next_temp1 = ttm_matrix;
                    next_state = Z_ROTATION;
                    next_theta = theta_z;
                    next_select = 2'b10;
                end
                Z_ROTATION:
                begin
                    next_temp0 = mt_matrix;
                    next_temp1 = ttm_matrix;
                    next_theta = theta_rotor;
                    next_select = 2'b10;
                    next_state = FINAL ;
                end
                FINAL :
                begin
                    next_matrix = mt_matrix;
                    next_rotor_matrix = ttm_matrix;
                    next_state = IDLE;
                end
            endcase
        end
    end
endmodule

module theta_to_matrix(theta, select, matrix);

    input [`TRIG_IN_BITS-1:0] theta;
    input [1:0] select;
    output [9*`TRIG_OUT_BITS-1:0] matrix;

    localparam MAX = {2'b01, {(`TRIG_OUT_BITS-2){1'b0}}};

    reg [`TRIG_OUT_BITS-1:0]
        a_x, a_y, a_z, b_x, b_y, b_z, c_x, c_y, c_z;
    assign matrix =
        {a_x, a_y, a_z, b_x, b_y, b_z, c_x, c_y, c_z};

    wire [`TRIG_OUT_BITS-1:0] cos, sin, nsin;
    wire [`TRIG_OUT_BITS:0] neg_sin;

    trig t (theta, sin, cos);
```

thetas_to_matrix.v

```
    signed_negate #(`TRIG_OUT_BITS) sn (sin, neg_sin);

    assign nsin = neg_sin[`TRIG_OUT_BITS-1:0];

    always @(theta, select, cos, sin, nsin) begin
        {a_x, a_y, a_z, b_x, b_y, b_z, c_x, c_y, c_z} =
            {(9*`TRIG_OUT_BITS){1'b0}};
        case (select)
            2'd0:
            begin
                a_x = MAX;
                b_y = cos;
                b_z = sin;
                c_y = nsin;
                c_z = cos;
            end
            2'd1:
            begin
                a_x = cos;
                a_z = nsin;
                b_y = MAX;
                c_x = sin;
                c_z = cos;
            end
            2'd2:
            begin
                a_x = cos;
                a_y = sin;
                b_x = nsin;
                b_y = cos;
                c_z = MAX;
            end
        endcase
    end
endmodule

module divider(reset, clock, enable);
    parameter COUNT = 10'd10;
    input clock;
    input reset;
    output enable;

    reg [9:0]counter;
    reg enable;

    always @(posedge clock)
        begin
            enable <= 1'b0;
            if (reset)
                counter <= 10'b0;
            else if (counter == COUNT)
                begin
                    counter <= 10'b0;
                    enable <= 1'b1;
                end
            else
                counter <= counter + 1;
        end
endmodule
```


labkit.v

```
///////////////////////////////////////////////////////////////////////////////
//
// 6.111 FPGA Labkit -- Template Toplevel Module for Lab 4 (Spring 2006)
//
//
// Created: March 13, 2006
// Author: Nathan Ickes
//
///////////////////////////////////////////////////////////////////////////////

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

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

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

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

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

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

           clock_feedback_out, clock_feedback_in,

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

           rs232_txd, rs232_rxd, rs232_rts, rs232_cts,

           mouse_clock, mouse_data, keyboard_clock, keyboard_data,

           clock_27mhz, clock1, clock2,

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

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

           switch,

           led,

           user1, user2, user3, user4,

           daughtercard,

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

           analyzer1_data, analyzer1_clock,
           analyzer2_data, analyzer2_clock,
           analyzer3_data, analyzer3_clock,
```

```
            analyzer4_data, analyzer4_clock);

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

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

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

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

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

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

   input  clock_feedback_in;
   output clock_feedback_out;

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

   output rs232_txd, rs232_rts;
   input  rs232_rxd, rs232_cts;

   input  mouse_clock, mouse_data, keyboard_clock, keyboard_data;

   input  clock_27mhz, clock1, clock2;

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

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

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

   inout [43:0] daughtercard;

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



```

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

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

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

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

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

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

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

   // 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;

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

   // User I/Os
```

labkit.v

```
   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;

   ////////////////////////////////////////////////////////////////////////////
   //
   // Lab 4 Components
   //
   ////////////////////////////////////////////////////////////////////////////

   wire pclk, pixel_clock, locked;

   assign pixel_clock = clock_27mhz;
   assign ram0_clk = pixel_clock;
   assign ram1_clk = pixel_clock;
   assign clock_feedback_out = 1'b0;

   DCM pixel_clock_dcm1 (.CLKIN(pixel_clock), .CLKFX(vga_out_pixel_clock));
   // synthesis attribute CLKFX_DIVIDE of pixel_clock_dcm1 is 2
   // synthesis attribute CLKFX_MULTIPLY of pixel_clock_dcm1 is 2
   // synthesis attribute CLK_FEEDBACK of pixel_clock_dcm1  is "NONE"
   // synthesis attribute CLKOUT_PHASE_SHIFT of pixel_clock_dcm1 is "NONE"
   // synthesis attribute PHASE_SHIFT of pixel_clock_dcm1 is 0

   // The composite sync signal is used to encode sync data in the green
   // channel analog voltage for older monitors.  It does not need to be
   // implemented for the monitors in the 6.111 lab, and can be left at 1'b1.
   assign vga_out_sync_b = 1'b1;

    wire reset_sync, up_sync, down_sync;
    wire [8:0] buttons;

    debounce db1 (1'b0, pixel_clock, ~button0, reset_sync);
    debounce db2 (1'b0, pixel_clock, ~button1, up_sync);
    debounce db3 (1'b0, pixel_clock, ~button2, down_sync);
    debounce db4 (1'b0, pixel_clock, ~button_right, buttons[0]);
    debounce db5 (1'b0, pixel_clock, ~button_left, buttons[1]);
    debounce db6 (1'b0, pixel_clock, ~button_down, buttons[2]);
    debounce db7 (1'b0, pixel_clock, ~button_up, buttons[3]);
    debounce db8 (1'b0, pixel_clock, ~button1, buttons[4]);
    debounce db9 (1'b0, pixel_clock, ~button2, buttons[5]);
    debounce db10 (1'b0, pixel_clock, ~button3, buttons[6]);
    debounce db11 (1'b0, pixel_clock, ~button_enter, buttons[7]);

    assign buttons[8] = switch[7];

    internal internal1(
        .reset(reset_sync), .clock(pixel_clock),

        .model_select({2'b0, switch[0], switch[6]}),

        .paddle_up(up_sync),
```


labkit.v

```
        .paddle_down(down_sync),
        .paddle_speed({switch[5:3], 1'b1}),
        .ball_initial_speed({switch[2:0], 1'b1}), // game inputs

        .ram0_data(ram0_data),
        .ram0_address(ram0_address),
        .ram0_we_b(ram0_we_b), // ram 0 buss
        .ram1_data(ram1_data),
        .ram1_address(ram1_address),
        .ram1_we_b(ram1_we_b), // ram 1 buss

        .mouse0_clock(mouse_clock),
        .mouse0_data(mouse_data), // ps/2 0 buss
        .mouse1_clock(keyboard_clock),
        .mouse1_data(keyboard_data), // ps/2 1 buss

        .buttons(buttons),

        .vga_rgb({vga_out_red, vga_out_green, vga_out_blue}),
        .vga_blank_b(vga_out_blank_b),
        .vga_hsync(vga_out_hsync),
        .vga_vsync(vga_out_vsync) // vga outputs
        );

    // SRAMs
    assign ram0_adv_ld = 1'b0;
    assign ram0_cen_b = 1'b0;
    assign ram0_ce_b = 1'b0;
    assign ram0_oe_b = 1'b0;
    assign ram0_bwe_b = 4'h0;
    assign ram1_adv_ld = 1'b0;
    assign ram1_cen_b = 1'b0;
    assign ram1_ce_b = 1'b0;
    assign ram1_oe_b = 1'b0;
    assign ram1_bwe_b = 4'h0;

    // Logic Analyzer
    assign analyzer1_data = 16'h0;
    assign analyzer1_clock = 1'b1;
    assign analyzer2_data =  {ram0_we_b, ram0_address[14:0]};
    assign analyzer2_clock = ram0_clk;
    assign analyzer3_data = 16'h0;
    assign analyzer3_clock = 1'b1;
    assign analyzer4_data = {ram0_data[15:0]};
    assign analyzer4_clock = ram0_clk;
endmodule

// Switch Debounce Module
// use your system clock for the clock input
// to produce a synchronous, debounced output
module debounce (reset, clock, noisy, clean);
   parameter DELAY = 270000;   // .01 sec with a 27Mhz clock
   input reset, clock, noisy;
   output clean;

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

   always @(posedge clock)
     if (reset)
       begin
      count <= 0;
      new <= noisy;
```

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

endmodule

```



```
`timescale 1ns / 1ps
module internal_tf_v;
    // Inputs
    reg reset;
    reg clock;
    reg [3:0] model_select;
    reg paddle_up;
    reg paddle_down;
    reg [3:0] paddle_speed;
    reg [3:0] ball_initial_speed;
    reg mouse0_clock;
    reg mouse0_data;
    reg mouse1_clock;
    reg mouse1_data;
    reg [7:0] buttons;
    reg [35:0] write_data;
    reg tristate;

    // Outputs
    wire [18:0] ram0_address;
    wire ram0_we_b;
    wire [18:0] ram1_address;
    wire ram1_we_b;
    wire [23:0] vga_rgb;
    wire vga_blank_b;
    wire vga_hsync;
    wire vga_vsync;

    // Bidirs
    wire [35:0] ram0_data;
    wire [35:0] ram1_data;

    assign ram0_data = (tristate) ? {36{1'bZ}} : write_data;

    // Instantiate the Unit Under Test (UUT)
    internal uut (
        .reset(reset),
        .clock(clock),
        .model_select(model_select),
        .paddle_up(paddle_up),
        .paddle_down(paddle_down),
        .paddle_speed(paddle_speed),
        .ball_initial_speed(ball_initial_speed),
        .ram0_data(ram0_data),
        .ram0_address(ram0_address),
        .ram0_we_b(ram0_we_b),
        .ram1_data(ram1_data),
        .ram1_address(ram1_address),
        .ram1_we_b(ram1_we_b),
        .mouse0_clock(mouse0_clock),
        .mouse0_data(mouse0_data),
        .mouse1_clock(mouse1_clock),
        .mouse1_data(mouse1_data),
        .vga_rgb(vga_rgb),
        .vga_blank_b(vga_blank_b),
        .vga_hsync(vga_hsync),
        .vga_vsync(vga_vsync),
        .buttons(buttons)
    );
    always #5 clock <= ~clock;

    initial begin
        // Initialize Inputs
```

internal_tf.v

```
        reset = 0;
        clock = 0;
        model_select = 0;
        paddle_up = 0;
        paddle_down = 0;
        paddle_speed = 0;
        ball_initial_speed = 0;
        mouse0_clock = 0;
        mouse0_data = 0;
        mouse1_clock = 0;
        mouse1_data = 0;
        tristate = 1;
        write_data = 0;
        buttons = 0;
        // Wait 100 ns for global reset to finish
        #100;
        reset = 1;
        @(posedge clock);
        @(posedge clock);
        @(posedge clock);
        reset = 0;
        @(posedge clock);
        @(posedge clock);
        @(posedge clock);
        tristate = 0;
        @(posedge clock);
        tristate = 1;
        @(posedge clock);
        tristate = 0;
        @(posedge clock);
        tristate = 1;
        @(posedge clock);
        tristate = 0;
        @(posedge clock);
        tristate = 1;
        @(posedge clock);
        tristate = 0;
        @(posedge clock);
        tristate = 1;
        @(posedge clock);
        tristate = 0;
        @(posedge clock);
        tristate = 1;
    end
endmodule
```


defines.v

```
`define COORD_BITS (12)
`define COLOR_BITS (24)
`define POINT_BITS (3 * `COORD_BITS)
`define TRIANGLE_BITS ((3 * `POINT_BITS) + `COLOR_BITS)
`define NORMAL_BITS (3 * `COORD_BITS)

// For Trig Functions
`define TRIG_IN_BITS (8)
`define TRIG_OUT_BITS (13)

// SVGA Output
`define PIXEL_BITS (10*2+12+24)
`define SCREEN_WIDTH (640)
`define SCREEN_HEIGHT (480)

`define P2D_BITS (2 * `COORD_BITS)
`define LINE_BITS (2 * `P2D_BITS)
`define RHOMBUS_BITS (2 * `LINE_BITS + 5*`COORD_BITS)

`define NUM_BUTTONS (9)

`define ROTOR_COLOR (24'h3F3FBE)
`define BALL_COLOR (24'hBEA33F)
`define PADDLE_COLOR (24'hBE3FBE)


`define MIT_RED  (24'h5B1F1F) //{8'b0101_1111, 8'b0001_1111, 8'b0001_1111})
`define MIT_GRAY  ({8'b0100_1111, 8'b0100_1111, 8'b0011_1111})
```

params.v

```
// params.v
// Global Constants (Used in more than one module)
parameter DISPLAY_WIDTH   = 10'd640 ;
parameter DISPLAY_HEIGHT    = 10'd480 ;
parameter PADDLE_WIDTH  = 10'd16 ;
parameter PADDLE_HEIGHT   = 10'd128 ;
parameter PADDLE_X  = 10'd28 ;
parameter BALL_WIDTH   = 10'd64 ;
parameter BALL_HEIGHT    = 10'd64 ;
parameter BORDER_WIDTH   = 10'd8 ;
parameter BORDER_HEIGHT    = 10'd8 ;
parameter MAX_SPEED   = 5'b11111 ;

parameter POS = 0;
parameter NEG = 1;
```

Page: 1