ECE 241 FINAL REPORT

Name of the project: "Punch Them All"
Team members: Mikael Del Castillo,1001125196
Luis Munoz Caicedo, 1001285147
Tutor: Moein (moein.ece@utoronto.ca)

INTRODUCTION

This engineering project consists of the development of the video game "Punch them all", which is a digital hardware application using knowledge from the ECE241 course at University of Toronto. The main goal of this project is to apply and learn hardware implementation skills to develop the game, by using: finite state machines, registers, to name a few verilog code components besides some pre-built hardware components. So, the game is able to generate graphic outputs and receive data from the player interaction. For the implementation, the *Altera DEs-1Soc* Field programmable Gate Array (FPGA) *board* is used to perform gaming calculations, display data output through a VGA adapter on a screen, and receive user input as feedback.

The game operates using verilog code, compiling and sending data to the FGPA, which displays images stored in RAM memory via the VGA screen, and respond according user input. The main characteristics of the game are a user-controllable puncher and self-moving zombies (targets). The puncher image moves vertically through a limited range and is controlled by the KEYS[3:2]. The targets appear at different locations at the right of the screen and move horizontally towards the left. Moreover, to emphasize the user interaction, when a KEY is pressed to hit a target, the puncher changes simulating a hitting animation. Every time the puncher hits a zombie, the score is updated and displayed on the 7 segment display. There is a limit of three lives per game, which are reduced every time the zombie pass or collides with the puncher without getting hit.

The game was developed focused on the fact that projects are important in engineering life, and this project was a good opportunity to enhance verilog coding skills, time management, and working in teams to accomplish the goal. The project was close to be fully implemented in the lapse of three weeks, with specific milestones. *See Appendix A.*

2. THE DESIGN

This project uses some pre-built hardware features of the FPGA to store and read from RAM memory, send data through a VGA output, and respond accordingly to the user input. Moreover, software components are used in form of verilog code to describe the circuit and the logic of the game. Both components combined work to accomplish the main functionality of the code, which is to display graphics stored in memory, and respond depending the user input to decide next steps. The hardware and software features will be explained below. For more information about the design, refer to Appendix B to see the verilog code, Appendix D to see the Block Diagram of the design and Appendix D to see the Game FSM.

Hardware Description:
The project uses hardware components from the De1-Soc board described below.

- **The push buttons**

The board provides four main push-button switches connected to the Cyclone V SoC FPGA: KEY0, KEY1, KEY2, and KEY3. Each one provides a high logic level when unpressed, and a low logic level otherwise. For this project, we inverted the functionality for KEY[3:0] to implement movement of the puncher and screen transitioning. At the press KEY3 and KEY2 the puncher will move up and down respectively and stop when the buttons are unpressed. KEY1 and KEY0 is used for hitting function and screen transitioning respectively.

- **The switches**

There are ten slide switches on the board, connected directly to a pin on the Cyclone V SoC FPGA. When the switch is in the down position (closest to the edge of the board), it provides a logic 0 to the FPGA, otherwise it provides a high logic one. This feature was used to implement the reset function using Switch 9 , which is activated at a logic 1.

- **The seven segment display**

The DE1-SoC board has six 7-segment displays (common anode. Each segment lights up applying a logic 0 and turn off at a logic 1. In this project all the 7-segment display are initialized to 0 and update every time the the puncher hits a target. The inputs for the 7-segment display are obtained from the output of the finite state machine described in the software section below.

- **The VGA adapter**

The DE1-SoC board includes a 15-pin D-SUB connector for VGA output. This output was connected to a screen for used feedback. In order to interface the VGA output with the output of the finite state machine, a VGA adapter library was used to implement graphic feedback.

- **The clock**

The DE1-SoC board has four 50 MHz clock sources from clock generator. In this project only one 50MHz clock is used to trigger functions in the verilog code and to built rate dividers to control the pace of certain functions. More information about how the clock was used is described in the software description section.

- **The RAM memory**

The board features 64MB of SDRAM, from which we used RAM: 1-PORT to store the game images. All the background, pucher, zombie and hit images were transformed to .mif files, and stored in memory for the use of the code. The design produces output from FPGA to fit a screen divided in 120x160 pixels. Each .mif file were used to create a 3 bits, 19200 words memory, where each bit represents one colour (either red, green or blue), and each word represents one pixel on the screen.

Software Description:
The project was partitioned in multiple modules that implemented together build the transitioning of the game. Each module will be called at the corresponding state of the FSM described below.

- **The Game FSM**

The game controller follow the state table described in Figure 1, where the start_key signal corresponds to the push of KEY0, delay is rate divider used to plot the images within a time window, and the game_over signal comes from the CHECK state in the datapath. *See Appendix C* to see image transitioning.

CURRENT STATE	NEXT STATE
HOME:	next_state = START;
START:	next_state = start_key ? START_WAIT : START;
START_WAIT:	next_state = start_key ? START_WAIT : BKGND;
BKGND:	next_state = delay ? PUNCHER : BKGND;
PUNCHER:	next_state = delay ? ZOMBIE : PUNCHER;
ZOMBIE:	next_state = delay ? HIT : ZOMBIE;
HIT:	next_state = delay ? CHECK : HIT;
CHECK:	next_state = game_over ? END : BKGND;
END:	next_state = start_key ? END_WAIT: END;
END_WAIT:	next_state = start_key ? END_WAIT: HOME;

Figure 1: State table

This FSM works together with the datapath which mainly consist of:

- **The VGA adapter code**

The VGA adapter takes a coordinate (X,Y) and the colour to be plot at that location. Each coordinate represents a pixel location on the screen. This code was used to print pixel by pixel images from memory on the screen, where each pixel color was represented by 3-bits.

- **The background drawing modules**

The outputs of this modules are a coordinate (X,Y) and the colour to be plot through the VGA adapter. The coordinates are obtained from two counters (X,Y) used to print every pixel in the 120x160 screen. The VGA translator is used to convert the X,Y coordinates into memory addresses and read colors from the memory allocated for the backgrounds .mif files. In this project, there are three background drawing modules, which change at the press of KEY1 when required. Each pixel is printed at the positive edge of the 50 MHz clock.

- **The puncher, zombie and heart_deducted drawing modules**

These modules use the same method as the background modules to read the colour from memory and display them on the screen. However, due to the fact that these images are smaller, the counters and drawing coordinates changed depending on the width and height of the object to be drawn and the location to be printed. The puncher module takes a Y coordinate as drawing start point. The Y coordinate is obtained from KEY3 AND KEY 4,

which shift the current position either up and down. The zombie module takes an X coordinate as drawing start point, which depends on a counter that move the image from right to left until a determined X coordinate. The heart_deducted module takes an X and Y input and print a black square on the top of the heart drawn in the background everytime the user lose a live.

- **The Counters**

Each graphic use a counter which will restart when reaching an specific point. The counters used to draw the images works with the positive edge of the 50 MHz clock while the speed of movement of the puncher and Zombie works with the positive edge of rate dividers of 1.25MHz for the Puncher and 1 MHz for the Zombie .

- **Collision**

The CHECK state in the datapath determines if the current location printed of the puncher matches the current location of the zombie. If it is the case, and no hit function is called, the heart-deducted drawing module is called. If the user lose three lives, the game over signal is activated.

- **The Hitting function**

This counter uses a rate divider to set the speed of the zombie. The hit module uses the last location that was printed on the screen and plot a modified puncher to simulate the animation.

- **Game over function**

The END is called when the game over signal from the datapath becomes logic 1. The END state uses the game over drawing module to plot the respective background and ask for a start button (KEY0) to go back to the START state.

3. REPORT ON SUCCESS

This project was built using multiple modules merged together using instances on the Top-Level entity verilog file. Each module was built and tested separately to ensure its functionality. We tested the drawing modules, and we were able to display the backgrounds, puncher, zombie and the black box on the top of the the heart background, see the screen transitioning on Appendix C. The puncher movement at the press KEY3 and KEY4 was achieved allowing vertical movement. The zombie moving feature also worked as the zombie appeared on the right most end of the screen, and moving towards the left until it reached an specific X coordinate of the screen.

Although this approach made easier to develop the transitioning, some difficulties were encountered as the modules were called on FSM. The main challenge we faced was the collision function. Our way to develop this function was to write a comparison using the location of the zombie and the puncher, after the hit state. If the X,Y of both modules coincide, it meant collision happened between a zombie and the puncher; and depending on the state of the puncher (hit or not hit) score was increased or lives were deducted. Even though, the code seems correct, lifes function was not being called until the third the

collision. So, by testing different locations of the puncher we found that the coordinates used to implement collision were not working as expected. We concluded that this may be happening because the counters used to plot the zombie and the puncher works differently due to the dimensions of the images.

Another challenge was the hit and score function, since both of them depended of the output of the collision function we were not able to test the functionality of this functions. Therefore, we left this sections and the hearth-deducted module incomplete.

4. WHAT WOULD WE DO DIFFERENTLY?

As mentioned before, the main issue arrived on the collision function. Because, some of the other functions and modules depended on the output of the collision function our projects was not working as planned. If the projects would be wrote from scratch again, we would:

- create less dependency between modules and functions returning the less variables as possible.
- save the coordinates of the last printed puncher and zombie in a bus every time the drawing modules were called for determining collision.

Other improvements would be changes in the organization of the project. As a matter of fact, planning diagrams. It is important to devote more time to diagrams and avoid jumping into coding directly without a clear idea of the steps to follow, which made debugging harder.

Moreover, we would ask more often for help to the TA in charge of the project. As students, we got easily stuck in steps where the advice of the TA would been more efficient in terms of time and functionality of the project.

APPENDIXES

Appendix A : Milestones of the project

The following milestones include, but are not limited to:

- November 14, 2016: Delivering of BMP2MIF files, including: background, targets, user aiming tool.
- November 21, 2016: Implementation of verilog code: movement of targets and aiming tools, implementation of score function, attach external inputs using Altera pre-built function codes (modules).
- November 28, 2016: Testing and delivering of a functional game.
- December 5 , 2016: Delivering of final project report.

Appendix B: All Verilog Code and Schematics

- **Top Entity Module:**

```
module punch_them_all
	(
		CLOCK_50,					// On Board 50 MHz
		// Your inputs and outputs here
		SW,
		KEY,
		// The ports below are for the VGA output.  Do not change.
		VGA_CLK,						// VGA Clock
		VGA_HS,						// VGA H_SYNC
		VGA_VS,						// VGA V_SYNC+
		VGA_BLANK_N,				// VGA BLANK
		VGA_SYNC_N,					// VGA SYNC
		VGA_R,						// VGA Red[9:0]
		VGA_G,						// VGA Green[9:0]
		VGA_B						// VGA Blue[9:0]
	);

	input		CLOCK_50;				// 50 MHz
	// Declare your inputs and outputs here
	input [9:0]SW;
	input [3:0]KEY;

	// Do not change the following outputs
	output			VGA_CLK;				// VGA Clock
	output			VGA_HS;					// VGA H_SYNC
	output			VGA_VS;					// VGA V_SYNC
	output			VGA_BLANK_N;			// VGA BLANK
	output			VGA_SYNC_N;				// VGA SYNC
	output	[9:0]	VGA_R;					// VGA Red[9:0]
	output	[9:0]	VGA_G;					// VGA Green[9:0]
	output	[9:0]	VGA_B;					// VGA Blue[9:0]

	wire resetn;
	assign resetn = SW[9];

	// Create the colour, x, y and writeEn wires that are inputs to the controller.

	wire [2:0] colour_vga;
	wire [7:0] x;
	wire [6:0] y;
	wire writeEn;
```

```
// image file (.MIF) for the controller.
vga_adapter VGA(
		.resetn(resetn),
		.clock(CLOCK_50),
		.colour(colour_vga),
		.x(x),
		.y(y),
		.plot(writeEn),
		/* Signals for the DAC to drive the monitor. */
		.VGA_R(VGA_R),
		.VGA_G(VGA_G),
		.VGA_B(VGA_B),
		.VGA_HS(VGA_HS),
		.VGA_VS(VGA_VS),
		.VGA_BLANK(VGA_BLANK_N),
		.VGA_SYNC(VGA_SYNC_N),
		.VGA_CLK(VGA_CLK));
	defparam VGA.RESOLUTION = "160x120";
	defparam VGA.MONOCHROME = "FALSE";
	defparam VGA.BITS_PER_COLOUR_CHANNEL = 1;
	defparam VGA.BACKGROUND_IMAGE = "screen.mif";

// Put your code here. Your code should produce signals x,y,colour and writeEn
// for the VGA controller, in addition to any other functionality your design may require

wire [2:0]transition;
wire gameover;

control C0(
	.clk(CLOCK_50),
	.resetn(resetn),
	.start_key(~KEY[1]),
	.gameover(gameover),

	.transition(transition)
	);
dpath (
	.clk(CLOCK_50),
	.resetn(resetn),
	.transition(transition),
	.up(~KEY[3]),
	.down(~KEY[2]),
	.gameover(gameover),

	.writeEn(writeEn),
	.x(x),
	.y(y),
	.colour_vga(colour_vga)
	);
```

```
endmodule

//======================================================================================================

module control(
   input clk,
   input resetn,
   input start_key,
   input gameover,

   output reg [2:0]transition
   //output reg setBlack
    );


    reg [3:0] current_state, next_state;

    localparam HOME         = 4'd0,
               START        = 4'd1,
               START_WAIT   = 4'd2,
               BKGND        = 4'd3,
               PUNCHER      = 4'd4,
               ZOMBIE       = 4'd5,
               CHECK        = 4'd6,
               END          = 4'd7,
               END_WAIT     = 4'd8;

   // Next state logic aka our state table
   always@(*)
   begin: state_table
      case (current_state)

          HOME:       next_state = START;
          START:      next_state = start_key ? START_WAIT : START;
          START_WAIT: next_state = start_key ? START_WAIT : BKGND;
          BKGND:      next_state = delay ? PUNCHER : BKGND;
          PUNCHER:    next_state = delay ? ZOMBIE : PUNCHER;
          ZOMBIE:     next_state = delay ? CHECK : ZOMBIE;
          CHECK:      next_state = gameover ? END : BKGND;
          END:        next_state = start_key ? END_WAIT: END;
          END_WAIT:   next_state = start_key ? END_WAIT: HOME;


      default:    next_state <= HOME;
          endcase

    end // state_table


    // Output logic aka all of our datapath control signals
   always @(*)
   begin: enable_signals
      // By default make all our signals 0

      case (current_state)
          HOME:       transition = 3'd0;
          START:      transition = 3'd1;
          BKGND:      transition = 3'd2;
          PUNCHER:    transition = 3'd3;
          ZOMBIE:     transition = 3'd4;
          CHECK:      transition = 3'd5;
          END:        transition = 3'd6;


        // default:    // don't need default since we already made sure all of our outputs were assigned a value at the start of the always block
        endcase
    end // enable_signals

    // current_state registers
    always@(posedge clk)
    begin: state_FFs
        if(!resetn)
            current_state <= HOME;
        else
            current_state <= next_state;
    end // state_FFS

    //delay for ploting
   wire delay;
   reg [23:0]count_delay;
   assign delay = (count_delay == 24'd200000)? 1 : 0;

   always @(posedge clk)begin
      if (delay)begin
         count_delay <= 24'd0;
      end
      else begin
         count_delay <= count_delay + 1'b1;
      end
   end

endmodule
```

```
//-------------------------------------------------------------------------------------------------
module dpath (clk, resetn, up, down, transition, gameover, writeEn, x, y, colour_vga);

   input clk, resetn;
   input [2:0]transition;
   input up, down;

   output reg gameover;
   output reg writeEn;
   output reg[7:0]x;
   output reg[6:0]y;
   output reg[2:0]colour_vga;


   //PLOTTING:

   wire[7:0]hx, sx, bx, px, zx, ex;
   wire[6:0]hy, sy, by, py, zy, ey;
   wire[2:0]hcolour, scolour, bcolour, pcolour, zcolour, ecolour;
   wire splot, pplot, zplot;

   // wire plot;
   always@(*)begin
      case(transition)
         3'd0: begin
            x = hx;
            y = hy;
            colour_vga = hcolour;
            writeEn = 1'b1;
         end
         3'd1: begin
            x = sx;
            y = sy;
            colour_vga = scolour;
            writeEn = 1'b1;
         end

         3'd2: begin
            x = bx;
            y = by;
            colour_vga = bcolour;
            writeEn = 1'b1;
         end

         3'd3: begin
            x = px;
            y = py;
            colour_vga = pcolour;
            writeEn = pplot;
         end

        3'd4: begin
           x = zx;
           y = zy;
           colour_vga = zcolour;
           writeEn = zplot;
        end

        3'd5: begin
           if((xreg == 10'd20) && (yreg < zy + 4'd10 || yreg > zy - 4'd10))
           begin
                 gameover = 1'b1;
           end
           else gameover = 1'b0;
        end

        3'd6: begin
           x = ex;
           y = ey;
           colour_vga = ecolour;
           writeEn = 1'b1;
        end

        default: begin
           x = ex;
           y = ey;
           colour_vga = ecolour;
           writeEn = 1'b0;

        end
    endcase
end



//DRAWING INSTANCES:

//HOME:
draw_home_screen h0(
.clk(clk),
.resetn(resetn),

.x(hx),
.y(hy),
.colour_out(hcolour)
);
```

```
    //START
    draw_start_key S0(
    .clk(clk),
    .resetn(resetn),
    .xPosition(10'd0),
    .yPosition(10'd500),

    .writeEn(splot),
    .x(sx),
    .y(sy),
    .colour_out(scoulour)
    );

    //PUNCHER
    draw_puncher P0(
        .clk(clk),
        .resetn(resetn),
        .y_in(yreg),

        .writeEn(pplot),
        .x(px),
        .y(py),
        .colour_out(pcolour)
        );

    //BKGND
    gameBKGND back0(
        .clk(clk),
        .resetn(resetn),

        .x(bx),
        .y(by),
        .colour_out(bcolour)
        );

    //ZOMBIE
    draw_zombie Z0(
        .clk(clk),
        .resetn(resetn),
        .x_in(xreg),

        .writeEn(zplot),
        .x(zx),
        .y(zy),
        .colour_out(zcolour)
        );


    //END
    draw_game_over G0(
        .clk(clk),
        .resetn(resetn),

        .x(ex),
        .y(ey),
        .colour_out(ecolour)
        );

wire RateDivider;
    reg [23:0]count1;
    reg [9:0]xreg;

    assign RateDivider = (count1 == 24'd1000000)? 1 : 0;

    always @(posedge clk)begin
        if (RateDivider)begin
            count1 <= 24'd0;
        end
        else begin
            count1 <= count1 + 1'b1;
        end
    end

    wire xcount;
    assign xcount = 1'b1;


    always@(posedge clk) begin
        if(!resetn)
            xreg <= 10'd790;
        else if(xreg == 10'd10)
            xreg <= 10'd790;

        else if (RateDivider) begin
                xreg <= xreg - xcount;
        end
    end


    wire RateDividerPuncher;
    reg [23:0]count2;
    assign RateDividerPuncher = (count2 == 24'd1250000)? 1 : 0;

    always @(posedge clk)begin
        if (RateDividerPuncher)begin
            count2 <= 24'd0;
            end
        else begin
            count2 <= count2 + 1'b1;
            end
    end

    wire ycount;
    assign ycount = 1'b1;
    reg [6:0]yreg;

    always@(posedge clk) begin
        if(!resetn)
            yreg <= 7'd0;

        else if (RateDividerPuncher) begin
            //colour_vga = pcolour;
            if (down)
                yreg <= yreg + ycount;

            if (up)
                yreg <= yreg - ycount;
        end
    end


endmodule
```

- **Puncher Verilog Code**

```
module draw_puncher(clk, resetn, y_in, writeEn, x, y, colour_out);

    input clk;
    input resetn;
    input [6:0]y_in;


    output reg writeEn;
    output [7:0]x;
    output [6:0]y;
    output [2:0]colour_out;

    wire load;

    wire plot;
    always@(*)begin
        if(colour_out == 3'b0)
            writeEn <= 1'b0;
        else
            writeEn <= plot;
    end


    puncher_control P_C0(
        .clk(clk),
        .resetn(resetn),

        .load(load),
        .plotOut(plot)
        );


    puncher_datapath P_D0(
        .clk(clk),
        .resetn(resetn),
        .yPosition(y_in),
        .load(load),

        .x(x),
        .y(y),
        .colour_out(colour_out)
        );

endmodule



module puncher_control(
    input clk,
    input resetn,
    output reg load, plotOut
    );
    reg [2:0] current_state, next_state;

    localparam      LOAD_Y          = 3'd0,
                    DRAW_PUNCHER    = 3'd1;

    // Next state logic aka our state table
    always@(*)
    begin: state_table
            case (current_state)
                LOAD_Y:         next_state = DRAW_PUNCHER;
                DRAW_PUNCHER:   next_state = LOAD_Y;
            default:    next_state = LOAD_Y;
        endcase
    end // state_table

    // Output logic aka all of our datapath control signals
    always @(*)
    begin: enable_signals
        // By default make all our signals 0
        load <= 1'b0;
        plotOut <= 1'b0;

        case (current_state)
            LOAD_Y: begin
                load <= 1'b1;
                end
            DRAW_PUNCHER: begin
                plotOut <= 1'b1;
                end
            // default:    // don't need default
            endcase
    end // enable_signals


     // current_state registers
    always@(posedge clk)
    begin: state_FFs
        if(!resetn)
            current_state <= LOAD_Y;
        else
            current_state <= next_state;
    end // state_FFS
endmodule
```

```
module puncher_datapath (clk, resetn, yPosition, load, x, y,colour_out);
    input clk;
    input resetn;
    input [6:0]yPosition;
    input load;

    output reg [7:0]x;
    output reg [6:0]y;
    output [2:0]colour_out;

    //COUNTERS
    reg [9:0] xCounter, yCounter;
    wire xCounter_clear;
    wire yCounter_clear;

always @(posedge clk or negedge resetn)
    begin
        if (!resetn)
            xCounter <= 10'd0;
        else if (xCounter_clear)
            xCounter <= 10'd0;
        else
        begin
            xCounter <= xCounter + 1'b1;
        end
    end
    assign xCounter_clear = (xCounter == (10'd130));

    /* A counter to scan vertically, indicating the row currently being drawn. */
    always @(posedge clk or negedge resetn)
    begin
        if (!resetn)
            yCounter <= 10'd0;
        else if (xCounter_clear && yCounter_clear)
            yCounter <= 10'd0;
        else if (xCounter_clear)       //Increment when x counter resets
            yCounter <= yCounter + 1'b1;
    end
    assign yCounter_clear = (yCounter == (10'd98));
```

```
    reg [6:0]yreg;
    reg [7:0]x_mem;
    reg [6:0]y_mem;
    //register X, Y
    always@(posedge clk) begin
       if(!resetn)
           yreg <= 7'b0;
       else begin
          if(load)
              yreg <= yPosition;
       end
    end

    always @(*)begin
          x <= xCounter[9:2];
          y <= yreg + yCounter[8:2];
          x_mem <= xCounter[9:2];
          y_mem <= yCounter[8:2];
    end

    //find memory adress to plot colour
    wire [14:0]mem_address;

    vga_address_translator M0(
    .x(x_mem),
    .y(y_mem),

    .mem_address(mem_address)
    );
    defparam M0.RESOLUTION = "160x120";

    //find colour
    puncherImage ima1 (
       .address(mem_address),
       .clock (clk),
       .data(0),
       .wren(0),
       .q(colour_out));

endmodule
```

- **Zombie Verilog Code**

```
odule draw_zombie(clk, resetn, x_in, writeEn, x, y, colo

   input clk;
   input resetn;
   input [7:0]x_in;


   output reg writeEn;
   output [7:0]x;
   output [6:0]y;
   output [2:0]colour_out;

   wire load;

   wire plot;
   always@(*)begin
       if(colour_out == 3'b0)
           writeEn <= 1'b0;
       else
           writeEn <= plot;
   end

   zombie_control P_C0(
       .clk(clk),
       .resetn(resetn),

       .load(load),
       .plotOut(plot)
       );


   zombie_datapath P_D0(
       .clk(clk),
       .resetn(resetn),
       .xPosition(x_in),
       .load(load),

       .x(x),
       .y(y),
       .colour_out(colour_out)
       );

ndmodule
```

```
module zombie_control(
   input clk,
   input resetn,
   output reg load, plotOut
    );
    reg [2:0] current_state, next_state;
    localparam     LOAD_X         = 3'd0,
                   DRAW_PUNCHER   = 3'd1;
     // Next state logic aka our state table
    always@(*)
    begin: state_table
            case (current_state)

               LOAD_X:         next_state = DRAW_PUNCHER;
               DRAW_PUNCHER:   next_state = LOAD_X;

            default:    next_state = LOAD_X;
        endcase
    end // state_table

    // Output logic aka all of our datapath control signals
    always @(*)
    begin: enable_signals
       // By default make all our signals 0
       load <= 1'b0;
       plotOut <= 1'b0;

       case (current_state)

          LOAD_X: begin
              load <= 1'b1;
              end

          DRAW_PUNCHER: begin
              plotOut <= 1'b1;
              end
         // default:    // don't need default since we already made sure all of our outputs were assigned a value at the start
        endcase
    end // enable_signals

     // current_state registers
    always@(posedge clk)
    begin: state_FFs
        if(!resetn)
            current_state <= LOAD_X;

        else
            current_state <= next_state;
    end // state_FFS
endmodule
```

```
module zombie_datapath (clk, resetn, xPosition, load, x, y,colour_out);
    input clk;
    input resetn;
    input [7:0]xPosition;
    input load;
    output reg [7:0]x;
    output reg [6:0]y;
    output [2:0]colour_out;

    //COUNTERS
    reg [9:0] xCounter, yCounter;
    wire xCounter_clear;
    wire yCounter_clear;

always @(posedge clk or negedge resetn)
    begin
        if (!resetn)
            xCounter <= 10'd0;
        else if (xCounter_clear)
            xCounter <= 10'd0;
        else
        begin
            xCounter <= xCounter + 1'b1;
        end
    end
    assign xCounter_clear = (xCounter == (10'd100));

    /* A counter to scan vertically, indicating the row currently being drawn. */
    always @(posedge clk or negedge resetn)
    begin
        if (!resetn)
            yCounter <= 10'd0;
        else if (xCounter_clear && yCounter_clear)
            yCounter <= 10'd0;
        else if (xCounter_clear)      //Increment when x counter resets
            yCounter <= yCounter + 1'b1;
    end
    assign yCounter_clear = (yCounter == (10'd95));


    reg [6:0]xreg;
    reg [7:0]x_mem;
    reg [6:0]y_mem;
    //register X, Y
    always@(posedge clk) begin
        if(!resetn)
            xreg <= 7'b0;
        else begin
            if(load)
                xreg <= xPosition;
        end
    end

    always @(*)begin
            x <= xreg +xCounter[9:2];
            y <= 7'd50 + yCounter[8:2];
            x_mem <= xCounter[9:2];
            y_mem <= yCounter[8:2];
    end

    //find memory adress to plot colour
    wire [14:0]mem_address;

    vga_address_translator M0(
    .x(x_mem),
    .y(y_mem),

    .mem_address(mem_address)
    );
    defparam M0.RESOLUTION = "160x120";

    //find colour
    zombieImage z0 (
        .address(mem_address),
        .clock (clk),
        .data(0),
        .wren(0),
        .q(colour_out));


endmodule
```

- **Drawing modules:**
- **Background**

```
module punch_them_all
    (
        CLOCK_50,                       // On Board 50 MHz
        // Your inputs and outputs here
        SW,
        KEY,
        // The ports below are for the VGA output.  Do not change.
        VGA_CLK,                        // VGA Clock
        VGA_HS,                         // VGA H_SYNC
        VGA_VS,                         // VGA V_SYNC+
        VGA_BLANK_N,                    // VGA BLANK
        VGA_SYNC_N,                     // VGA SYNC
        VGA_R,                          // VGA Red[9:0]
        VGA_G,                          // VGA Green[9:0]
        VGA_B                           // VGA Blue[9:0]
    );

    input       CLOCK_50;               // 50 MHz
    // Declare your inputs and outputs here
    input [9:0]SW;
    input [3:0]KEY;

    // Do not change the following outputs
    output          VGA_CLK;            // VGA Clock
    output          VGA_HS;             // VGA H_SYNC
    output          VGA_VS;             // VGA V_SYNC
    output          VGA_BLANK_N;        // VGA BLANK
    output          VGA_SYNC_N;         // VGA SYNC
    output   [9:0] VGA_R;               // VGA Red[9:0]
    output   [9:0] VGA_G;               // VGA Green[9:0]
    output   [9:0] VGA_B;               // VGA Blue[9:0]

    wire resetn;
    assign resetn = KEY[0];

    // Create the colour, x, y and writeEn wires that are inputs to the controller.

    wire [2:0] colour_vga;
    wire [7:0] x;
    wire [6:0] y;
    wire writeEn;


    // Create an Instance of a VGA controller - there can be only one!
    // Define the number of colours as well as the initial background
    // image file (.MIF) for the controller.
    vga_adapter VGA(
            .resetn(resetn),
            .clock(CLOCK_50),
            .colour(colour_vga),
            .x(x),
            .y(y),
            .plot(writeEn),
            /* Signals for the DAC to drive the monitor. */
            .VGA_R(VGA_R),
            .VGA_G(VGA_G),
            .VGA_B(VGA_B),
            .VGA_HS(VGA_HS),
            .VGA_VS(VGA_VS),
            .VGA_BLANK(VGA_BLANK_N),
            .VGA_SYNC(VGA_SYNC_N),
            .VGA_CLK(VGA_CLK));
        defparam VGA.RESOLUTION = "160x120";
        defparam VGA.MONOCHROME = "FALSE";
        defparam VGA.BITS_PER_COLOUR_CHANNEL = 1;
        defparam VGA.BACKGROUND_IMAGE = "screen.mif";

    // Put your code here. Your code should produce signals x,y,colour and writeEn
    // for the VGA controller, in addition to any other functionality your design may

    wire [2:0]transition;
    wire gameover;

    control C0(
        .clk(CLOCK_50),
        .resetn(resetn),
        .start_key(~KEY[1]),
        .gameover(gameover),

        .transition(transition)
        );
    dpath d(
        .clk(CLOCK_50),
        .resetn(resetn),
        .transition(transition),
        .up(~KEY[3]),
        .down(~KEY[2]),
        .gameover(gameover),

        .writeEn(writeEn),
        .x(x),
        .y(y),
        .colour_vga(colour_vga)
        );
endmodule
```

- **Home Screen and Gave Over screen:**

Both modules use the same code as for background, only changing the instances of the memory to call.

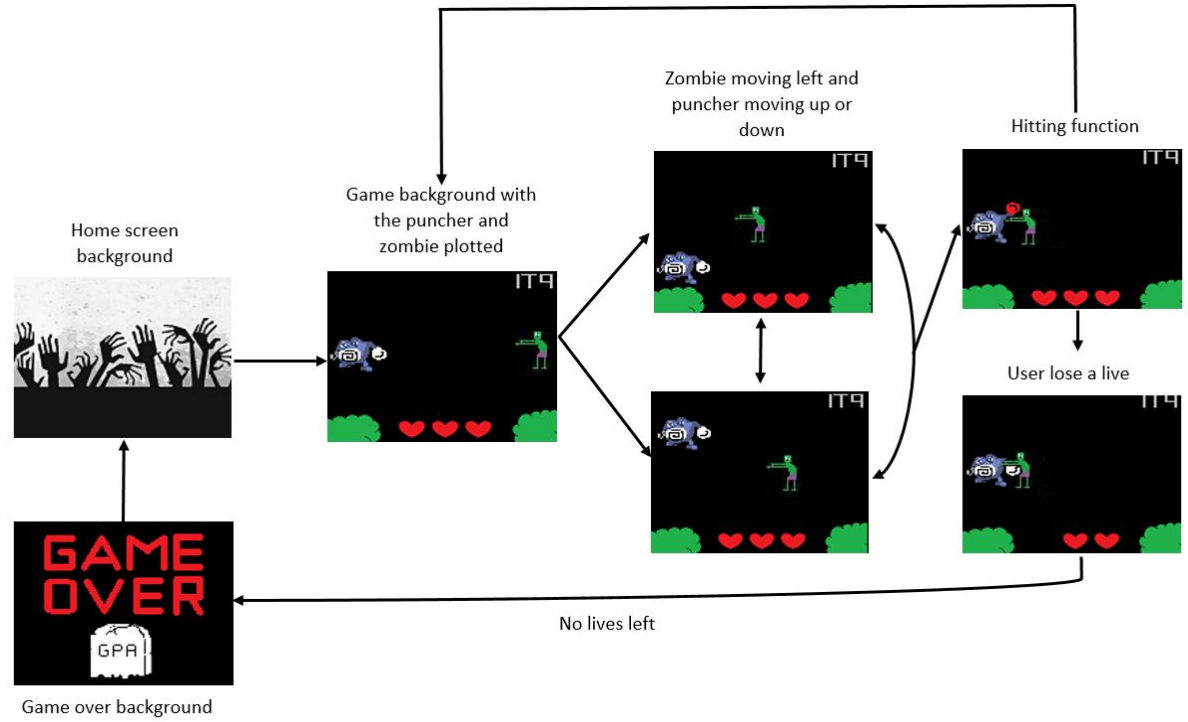
```
    home_screen ima1 (
        .address(mem_address),
        .clock (clk),
        .data(0),
        .wren(0),
        .q(colour_out));

endmodule
```

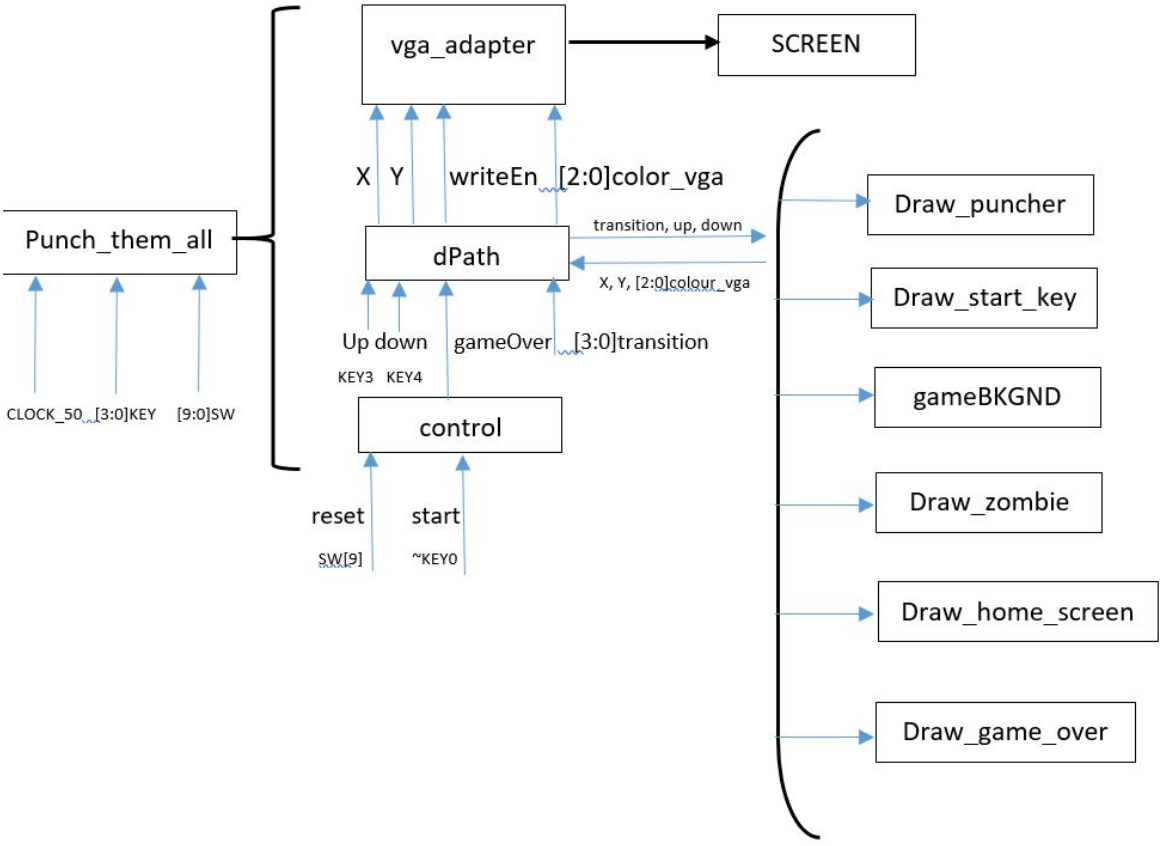
```
    gameOver ima1 (
        .address(mem_address),
        .clock (clk),
        .data(0),
        .wren(0),
        .q(colour_out));

endmodule
```

Appendix C: Image Transition



Appendix D: Image Block Diagram



Appendix E: Game FSM