FPGA Light Zapper

By Griffin Duffy and Aiden Padilla

Table of Contents

Overview

Our project is based around the idea of a light gun, a type of input device that utilizes a light sensor and can detect a specific target displayed on a screen by looking for changes in brightness on the display. This type of input device was first developed at MIT in the mid-20th century in order to interact with the CRT displays of the time and be a regular computer input device, however the technology became much more popular with the rise of arcades and home gaming systems such as the NES. In these applications, light gun technology became and exciting way for game developers have their players interact with their games.

For our game, we have taken aspects of both Nintendo's Duck Hunt and Fruit Ninja to create our own game in which targets are thrown upwards from the bottom of the screen and it's the player's goal to hit as many of the targets as possible before they drop using the light gun. In addition to this basic mode, we've added an additional game mode and several difficulties that players can either choose from or the game will naturally progress to.

The undertaking of this project has been very interesting since it contains not only the logic portion on the FPGA, but also the analog component of the zapper itself and the phototransistor it's built around as well.

Motivation

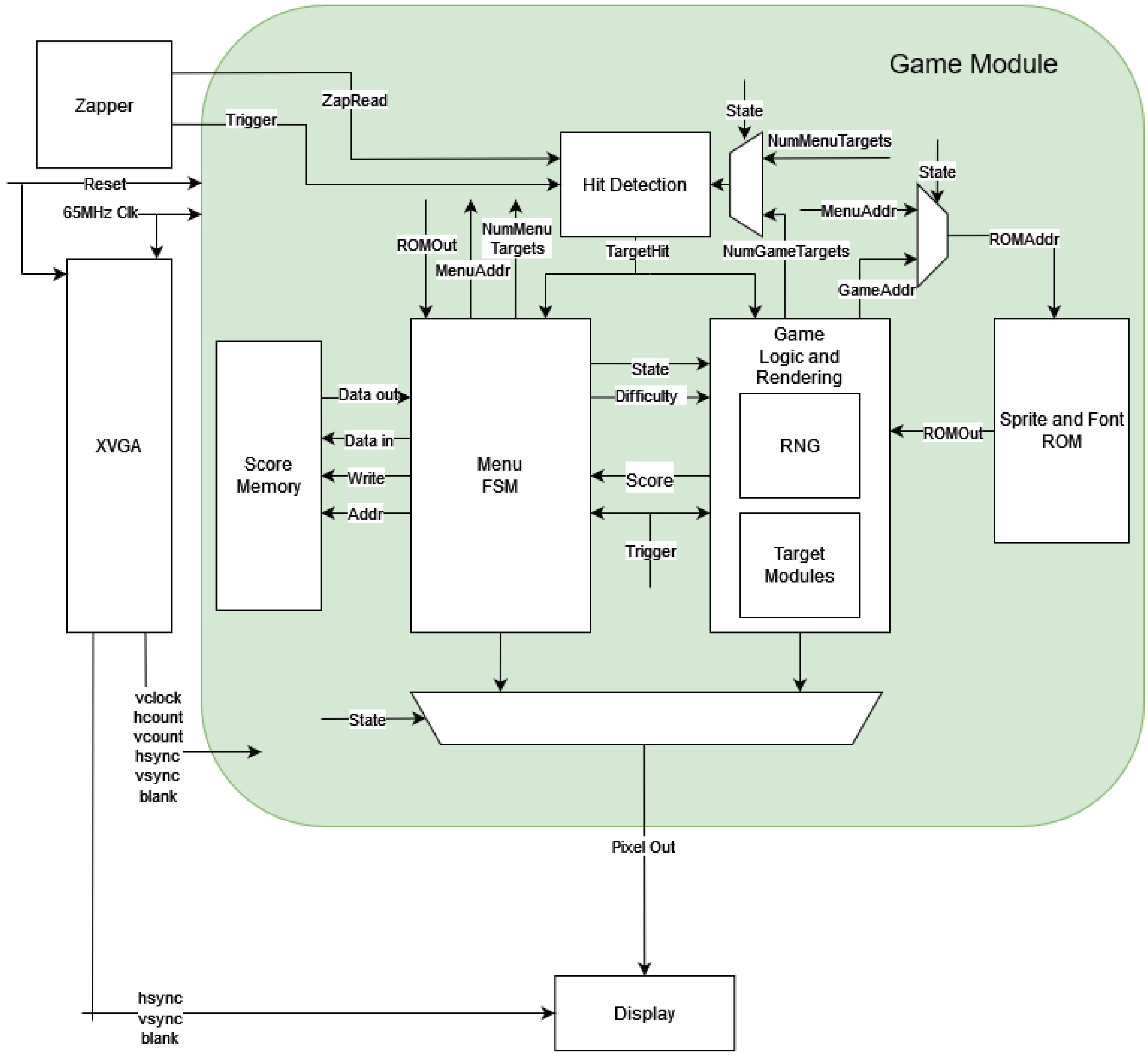
The motivation for this project was primarily due to two aspects: the desire to make something both fun and visually interesting, and the feasibility of the project. On the subject of the feasibility, we already had some experience in the labs working with video outputs and making a game (albeit a rudimentary one) in the form of pong. Changing pong to be targets going up and down did not seem to be too much of a logical leap. Additionally, light gun technology is decades old; the NES Zapper was released in 1984, thirty seven years ago. It was our belief that if the technology was feasible thirty seven years ago then it should theoretically present little problem for us to create today with the tools at our disposal.

Second was the desire to create something both fun and visually interesting. Of course, ideally all projects one chooses to work on should be interesting to that person, but more than just being "interesting", it was nice to work on something which was actively fun and became more fun the more features we added; it acted as a sort of incentive. Frequently while debugging or waiting for bitstreams to compile we would spend time just playing the game and seeing the highest score we could get. This also ties into the desire to create something visually interesting. There are interesting projects one could do based almost entirely around manipulation of data which have almost no visual aspect to them, but these require a lot of explanation to understand what's going on. Creating a light gun and a corresponding game is not only visually interesting, it speaks for itself when it comes to what the project is. Anyone who looks at our finished product can immediately understand what it is and how it works practically with no explanation from us, which we see as a benefit.

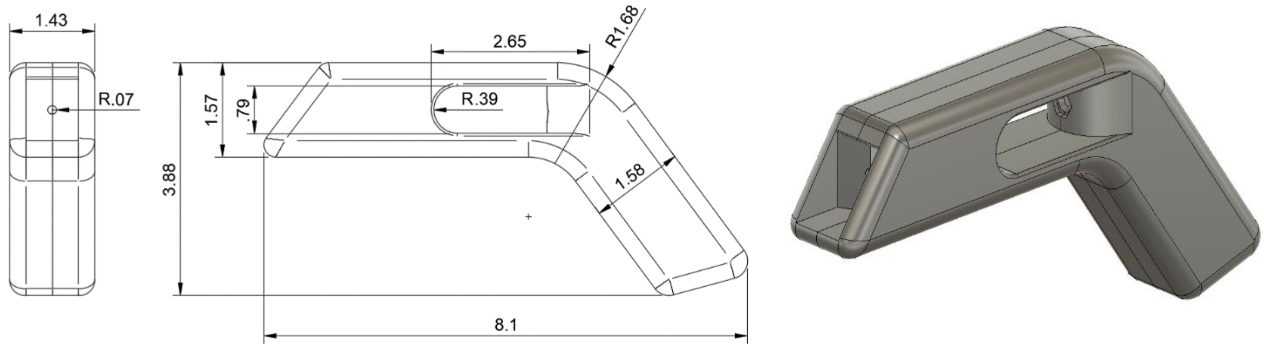
Summary

For our project, we chose to implement a Duck Hunt and Fruit Ninja hybrid game in which players use a light gun to hit fruit as they are thrown up and fly across the screen. Players can choose between different difficulties and game modes from the main menu which also utilizes the light gun and their score is reported to them on the FPGA’s seven segment display.

Block Diagram



Light Zapper



At its core, the light zapper consists of a phototransistor with its collector pulled-up high to VCC and its emitter is kept at ground such that when enough light hits the phototransistor, the switch of the transistor is closed and we can measure the collector drop from VCC to ground and thus an active low signal is established. Additionally, the light gun also houses the trigger which the player pulls to “shoot” the light gun at the screen and begin the hit detection process. While the analog circuitry is the main functional component of the light zapper, our team also decided to CAD an enclosure for or circuitry in order to block out ambient light from hitting the phototransistor and potentially triggering it when not receiving enough light from the display; as an added benefit, creating the zapper enclosure makes the game much more exciting for the player than simply holding some random components on a breadboard. Once our model was printed out, we were able to solder together the necessary circuitry, secure it in the enclosure, and connect it to the FPGA via cables coming out of the bottom.

Our printed design fitted with the necessary hardware and wiring

A look at the internal components and wiring

Modules

Top Level

Our Top Level module is responsible for controlling all I/O for our implementation as well as making all necessary nets and connections between all lower modules. In terms of I/O, the largest consideration for our project was being able to output via VGA to a monitor to have a working game; as a result, the entirety of our system operates with a 65mhz clock to accommodate the timing requirements of 1024x768 XVGA resolution. Additionally, the Top Level module is where the signals for the light zapper are received such that the Top Level module can then route them to each other module as needed. While not entirely necessary to be within Top Level, players' scores are calculated here and sent directly to the seven-segment display. Finally, the Top Level is also responsible for assigning the final pixel output, displaying nonzero Game FSM output pixels and otherwise replacing entirely black pixels with background pixels from the Background ROM.

Game FSM

The Game FSM or Menu FSM controls the main menu behavior and acts as a “controller” for the actual game itself and intermediary between the game and other systems such as hit detection. Hit detection information is passed in, and pixel information to be displayed is passed out. The Menu FSM consists of a state machine with various states for navigation of the menu as well as for gameplay.

Beginning in the very main menu, users can traverse to the pre-game menu where settings can be selected: whether the difficulty starts off as low or high, as well as the game mode they would like to select (either a normal game or the “keep-up” game). This is accomplished through the same target hit detection which works while in-game, only applied to menu buttons instead of targets themselves. All targets are loaded in at all times, but which ones are active and thus visible and available for hit detection is controlled by targets_activated and targets_onscreen variables.

Upon starting a game, the Menu FSM begins a counter which is incremented every cycle. Upon reaching one minute of real time, the game is over, and the user is returned from the game into the main menu.

Additionally, every third of the total game time (which is every twenty seconds given a one minute game), the difficulty increases by one. Functionally, what this means is that when starting at difficulty zero, targets will be thrown and fall slowly for the first twenty seconds, be thrown higher and fall faster for the next twenty, and require two hits to break for the final twenty. If starting on difficulty one instead of zero, targets start out already moving faster and take two hits for the final forty seconds of game time.
If selecting the “keep-up” game mode, difficulty again controls the speed of the target and gravity, with difficulty zero having a floatier and slower target and difficulty one’s target moving much faster.

For the basic gameplay mode, we also were able to utilize the RNG module to randomize the spawns of the targets being thrown up on the screen. Whenever there is less than two targets active and onscreen, the Game FSM module will take the random output of the RNG module and use it to select different targets by basing it off of a certain threshold value. If the randomly generated number was lower than that threshold value, that target would be selected. Therefore, we could adjust probabilities of each target accordingly based on their point worth by changing these threshold values.

Hit Detect

The hit detect module consists of a Moore finite state machine with six individual states consisting of the following: `IDLE`, `TRIGGERED`, `TARGET_CHECK`, `HIT`, `MISS`, `ASSERT`. In the `IDLE` state, once a trigger press has been registered, there are targets present on screen, and the display is about to begin the next frame, it will transition to the TRIGGERED state and otherwise stay in the IDLE state. Upon transitioning into the TRIGGERED state, the module will store a one-hot-encoded signal of onscreen targets in a register and reset various counters in preparation for the next stage. Once in the TRIGGERED state, the module will determine the indices of which targets need to be flashed and store those values in the index_to_flash array for later use in the TARGET_CHECK, HIT, and MISS states—it then transitions into the TARGET_CHECK state after the end of the initial blank frame. From this point, the FSM will continuously move between the TARGET_CHECK state and the MISS state until a hit is registered by a rising edge of zap_in or all targets onscreen have been flashed. If a hit is registered, our module transitions to the HIT state, asserts its output is valid along with setting target_hit to the corresponding target in a one-hot encoding scheme. Upon a frame ending and a hit not being registered, the module will move to the MISS state where it prepares to flash the next target in index_to_flash. If there are no more targets to flash, it moves to ASSERT, asserting that no target has been hit. Finally, the ASSERT state resets important output registers and then returns to IDLE.

Game Targets

The Game Targets module is responsible for controlling the motion of the targets which appear during gameplay. It functions as a state machine with three states: VISIBLE, MISSED, and NONVISIBLE. Entering the VISIBLE state, the target begins with an initial horizontal, vertical velocity, and initial horizontal position all determined by the Random Number Generator. Targets with an initial horizontal position on the left side of the screen fly from left to right and vice versa, so as to eliminate situations where a target begins near a corner and immediately flies off screen. The target begins with its vertical position at the maximum (i.e. off the bottom of the screen) and begins flying upwards. Each frame, the target's horizontal and vertical positions are translated via its horizontal and vertical velocity, and the target's vertical velocity is reduced by a constant parameter of GRAVITY. In this way, the target rises and falls in a realistic and satisfying manner, as if determined by the way that real objects fall via gravity. The value of GRAVITY can be altered based upon the difficulty level passed into the Game Targets module, such that on higher difficulties the level of gravity is increased and targets will fall faster, making them harder to hit as they move faster and spend less time on screen. When the target reaches the apex of its trajectory (i.e. the value of its vertical velocity is either 0 or is less than the amount that it would be decreased by GRAVITY), its vertical direction goes from UP to DOWN. When this happens, every frame we now *add* GRAVITY to the target's vertical velocity which is now *de facto* negative. We do this to simplify our code and avoid having to work with signed negative numbers, which could lead to bugs.

When the target has flown up and then reaches the bottom of the screen again without having been hit, the player has failed to hit it and we move into the MISSED state. This is a primarily transitory state which we spend only a single frame in before moving into the NONVISIBLE state. In the NONVISIBLE state, the target waits a random amount of frames from 0 to 30 (i.e. 0 to 0.5 seconds at 60 frames per second) before reappearing and moving back into the VISIBLE state. We do this for the sake of better gameplay, as it becomes overwhelming if targets were to immediately reappear after being hit or being missed; the player needs time to visually confirm whether a target has been hit or missed before it goes up again. Before transitioning back into the VISIBLE state, we again call the Random Number Generator to provide the target with new initial velocities and an initial horizontal position.

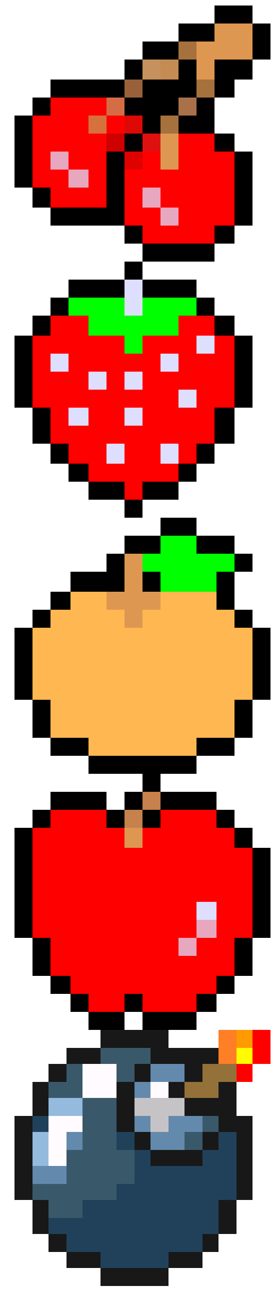
All aforementioned Game Target updates take place once every frame. Every clock cycle, however, we check to see if the target has been hit, i.e. if the input *is_hit* to the target is one. If so, the target immediately transitions to the NONVISIBLE state on the next frame, where its vertical position is set to the height of the screen such that it is not visible, and appears to the user as if the target was hit and destroyed.

If the difficulty level is three when the target is hit, however, a separate behavior occurs. If this is the first time that the target has been hit since entering the VISIBLE state, it changes its horizontal direction from left to right or vice versa and gains a small amount of upwards vertical velocity, in effect “hopping up” slightly while changing direction. The second time the target is hit, it is “destroyed” as normal. This behavior provides an extra challenge when the game is in the later stages of difficulty, as targets take multiple hits and their movement changes suddenly.

There is an alternative Game Target module called Popup Target, meant for the second game mode. In this game mode, rather than multiple targets which fly up on screen and are destroyed as the player hits them, there is a single target which exhibits the “hopping up” or “bouncing” behavior every time it is hit. The aim of the game mode is to keep the target up on screen as long as possible by hitting it continuously such that it never falls down. This Popup Target’s behavior is mostly identical to a regular Game Target’s behavior when difficulty equals three, with the only difference being that the Popup Target’s horizontal velocity changes randomly upon being hit, along with the aforementioned property that it is not destroyed after any number of hits (unlike a regular target at difficulty three, which is destroyed after two hits). To summarize, when a Popup Target is hit, its horizontal direction reverses, it begins moving upwards if it was falling down, gains a small constant amount of vertical velocity, and gets a new randomly determined amount of horizontal velocity.

Sprite ROM

Within our Sprite ROM, we are store all five of our fruit and bomb sprites, each being 128x128 in size and all using the same set of RGB colormaps. Thus, with 8-bit colors and colormap addresses and 4860kbits of BRAM available on the FPGA, we have $\frac{128*128*5*8+3*256*8}{4860*1000} = \sim 13.6\%$ memory utilization. If there were more sprites we wanted to use or need for additional BRAM to elsewhere, we could have scaled down our sprites here and simply upscaled when displaying but we deemed such to be unnecessary for our project. For the Sprite ROM module itself, it will lookup the appropriate sprite based on the object being currently drawn in the game logic and output it on the third cycle after the input due to some pipelining with the address calculation; resultingly, it is given an offset `hcount_in` input to ensure the sprites are output cleanly. In order to simplify our logic, the sprites are also arranged vertically such that we only require a vertical offset when looking up into the ROM and not a horizontal one as well which would be the case if say there were three in the first row and the remaining two in a second row. An additional consideration that had to be made within this module is the fact that the Top Level module replaces all black pixels with the background image as well as the white backgrounds of the sprites.



In order to address this and still keep the black border pixels of the sprites, our solution was to simply flip the least significant bit of any output black pixel to prevent the swap from occurring in Top Level as well as turn any entirely white pixel to entirely black to make sure it gets replaced with the appropriate background.

Background ROM

The Background ROM module is quite similar to the Sprite ROM module, with the difference being there is some upscaling and wrapping of the image at play here. Instead of using a 1024x768 image for it to be able to cover the entirety of the display, we instead used a 256x384 image which was then used to entirely cover the bottom half of the display by taking the `hcount_in % 256` for the input to address calculation so that we would be able to draw the background in a repeated fashion. Additionally, since the top half of the original image was simply blue anyways, we copied the same blue value from the COE file to be written as the background pixel for every value below the 384 halfway threshold of the display height.

Random Number Generator

In a game like the one we wanted to make, it's crucial that the challenge that players face is dynamic. It would be uninteresting and quickly get stale if the targets appeared in the same pattern, in the same position, and with the same velocities every time the game were played. Thus, it becomes necessary for us to introduce elements of randomness into our game, which requires a Random Number Generator.

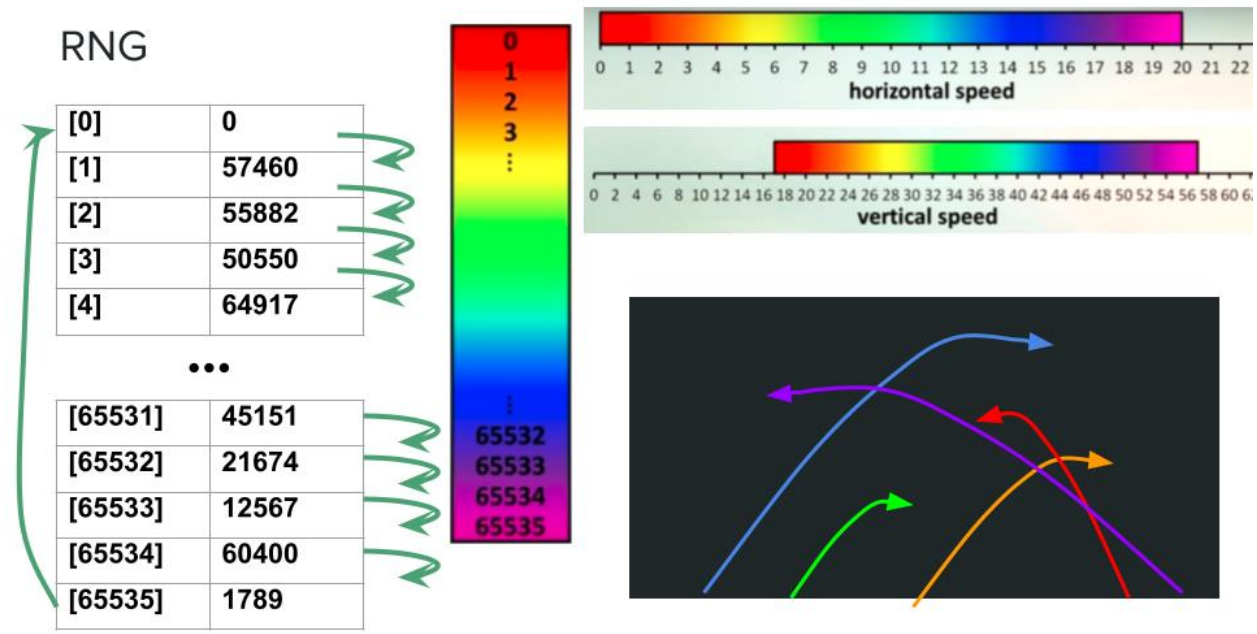
Generating random numbers completely deterministically and on an FPGA is less difficult than one might expect. It helps that the type of randomness we need doesn't need to be anywhere near as truly random or as cryptographically secure as many other computational tasks.

All randomness is generated through a single 16 bit RNG variable. When something like a target wants randomness, it calls the RNG module. The RNG module takes the current RNG state value as input, generates a new RNG value, and outputs it. That output value is then stored as the next RNG state value.

The actual process of transitioning from one RNG state to the next is best described as "bit magic". We first take our current state, shift it to the right by 5 bits, and XOR it with itself, storing this value as "temp1". We take temp1, shift it to the right by 7 bits, and XOR it with itself, storing this as temp2. Finally, our new RNG state is temp2 plus decimal 28383, and we set our RNG output value to this.

Given that our RNG state is a 16 bit variable, there are 65,536 possible inputs and the same number of possible outputs. Our RNG function is a bijection, meaning that it maps to every output, and no two inputs map to the same output. It forms a cycle 65536 long before repeating back on itself: if you were to call RNG 65536 times you would go through every value before ending back at your original state.

Once an RNG value has been returned to whatever required it, working with that value to produce the desired random behavior is simple. If the goal is to do something with a certain probability *x*, we can simply check to see if the return value is less than *x * 65536*, which we expect to happen x% of the time. Or, as we frequently need to do in the target module to give targets randomly determined initial velocities, we may want to get a range of possible random numbers. To do this, we can take our returned RNG value modulo whatever we desire our range to be. So, if I wanted a target's possible initial velocity to range from 3 to 15, I would set it to be equal to 2 + (RNG % 13 + 1), which again we expect to be distributed uniformly across that range.



Challenges

One of the largest challenges we faced was unfortunately also the most crucial aspect of our design, being able to interface with the light zapper hardware and accurately detect a hit and which target was hit. Another issue that we did not realize until during our final checkoff was an oversight in the light gun circuitry in which we actually had a floating input from our trigger. This went unnoticed since we had switched out our trigger button at the last moment before our checkoff and during the previous rounds of testing, a different button had been used that luckily (or perhaps unluckily for us) worked regardless potentially due to collecting greater static charge. Another significant challenge we ran into over the course of our project was communication, there were many times when we would both be making edits to a single file and by the time one of us had pushed to git, we would have to manually resolve the differences—wasting time. Time management was also another struggle for us; although we were able to meet all of our goals pretty handedly before Thanksgiving break, it afterwards became a struggle and we ended up missing several of our commitment goals.

Improvements

One improvement that can be made would be the addition of text to our menus and during gameplay. This could be achieved by initializing an additional ROM file containing a font and another `text_box` module that would take a string of characters as an input as well as an anchoring corner and then drawing the given characters from the point of the given corner. This would greatly enhance the usability of our project since it would enable players to know what each button in the onscreen menus does. Further improvements could also be made to our hit detection algorithm; instead of implementing a method that involves individually flashing each target one-by-one resulting in a number of required frames being linear with the number of onscreen targets. Instead, we could have successfully implemented a method that first flashes all targets to check if one is hit and then if so, doing a binary search on the targets to figure out exactly which target was hit resulting in the number of required frames for hit detection to be 1 + log2(n).

Conclusions

Overall, we consider this project a success. Despite the limitations of our hardware, we accomplished the goal we set out in creating a light gun game. Given the challenges of having to interface with the physical world as opposed to operating completely on the FPGA, we still ended up with a quite functional light gun. Additionally, despite the seeming simplicity of our game, it turned out as surprisingly fun—there is something inherently satisfying about clicking away fruits just as they jump up, or in keeping an apple bouncing left to right for as long as you can keep hitting it. We ran into issues of working with our hardware and time concerns, but ended up implementing all of the most important features that we initially set out to.

Appendix

```
`default_nettype none


module top_level(

    input wire clk_100mhz,

    input wire [15:0] sw,

    input wire btnc, btnu, btnl, btnr, btnd,

    input wire [1:0] jc,

    output logic [3:0] vga_r, vga_g, vga_b,

    output logic vga_hs, vga_vs,

    output logic [15:0] led,

    output logic ca, cb, cc, cd, ce, cf, cg, dp,

    output logic [7:0] an

    );

    localparam  TARGET_WIDTH = 64;
```

```
    logic reset;
    assign reset = btnc;

    // clock manager
    logic clk_65mhz;
    clk_wiz_0   (.clk_in1(clk_100mhz), .clk_out1(clk_65mhz));

    // test trigger for debug
    logic prev_trig, trig_rise, trig_read;
    assign trig_rise = ~prev_trig && trig_read;

debounce(.reset_in(reset), .clock_in(clk_65mhz), .noisy_in(trig_
read_noisy), .clean_out(trig_read));

    // seven seg display
    logic [6:0] seg_out;
    assign {cg, cf, ce, cd, cc, cb, ca} = seg_out;
    seven_seg_controller
sev_seg(.clk_in(clk_100mhz), .rst_in(reset), .val_in(seven_seg_i
nput), .cat_out(seg_out), .an_out(an));

    logic [31:0] seven_seg_input;
    assign seven_seg_input = {game_timer, 3'b000, hit_count};
    logic [11:0] game_timer;



    // display logic
```

```
    logic [11:0] pixel;
    logic [11:0] hit_detection_pixel;
    logic [10:0] hcount, prev_hcount;
    logic [9:0]  vcount, prev_vcount;
    logic hsync, vsync, blank;

    display
(.clk_65mhz(clk_65mhz), .reset(reset), .pixel(pixel), .vga_r(vga
_r),
                    .vga_g(vga_g), .vga_b(vga_b), .hcount(hcount
),
                    .vcount(vcount), .hsync(hsync), .vsync(vsync
),
                    .blank(blank), .vga_hsync(vga_hs), .vga_vsyn
c(vga_vs));

    // background display
    logic [11:0] bg_out;
    background_rom
bg(.pixel_clk_in(clk_65mhz), .hcount_in(hcount), .vcount_in(vcou
nt), .pixel_out(bg_out));

    // target logic
    //logic [11:0] output_pixel
    logic [3:0] difficulty;
    logic [11:0] game_out, target_flash, targets_onscreen,
hit_target;
```

```
    game_fsm
my_game_fsm(.clk_65mhz(clk_65mhz), .reset_in(reset),
                    .hcount_in(hcount), .vcount_in(vcount), .tri
gger_in(trig_rise),
                    .active_in(pause), .flash_in(target_flash),
                    .hit_valid(hit_valid), .hit_target(hit_targe
t),
                    .left_button_in(btnl), .right_button_in(btnr
), .down_button_in(btnd),
                    .pixel_out(game_out), .difficulty(difficulty
),
                    .targets_onscreen(targets_onscreen), .timer_
out(game_timer),
                    .new_game_out(new_game));
    assign led[11:0] = targets_onscreen;

    // pixel output
    assign pixel = |game_out ? game_out : bg_out;

    // hit detection
    logic hit_valid, targets_active, pause;
    assign pause = sw[15] ? sw[14] : targets_active;
    hit_detect
hd(.clk(clk_65mhz), .reset(reset), .zap_in(zap_read), .trig_in(t
rig_rise),
                    .targets_onscreen(targets_onscreen), .hcount
_in(hcount), .vcount_in(vcount),
```

```
                    .targets_update(targets_active), .flash_out(
target_flash),
                    .valid_out(hit_valid), .target_hit(hit_targe
t));

    logic [15:0] hit_count;
    logic new_game;
    always_ff @(posedge clk_65mhz) begin
        if (reset) begin
            prev_trig <= 1'b1;
            hit_count <= 0;
        end else begin
            prev_trig <= trig_read;
            hit_count <= new_game ? 0 : (hit_valid ?
hit_target : hit_count);
            // score increments
            if(hit_valid) begin
                case(hit_target[11:7])
                    5'b00001: hit_count <= hit_count + 10; //
cherry
                    5'b00010: hit_count <= hit_count + 15; //
strawberry
                    5'b01000: hit_count <= hit_count + 25; //
orange
                    5'b10000: hit_count <= hit_count + 50; //
apple
```

```
                    5'b00100: hit_count <= hit_count > 50 ?
hit_count - 50 : 0; // bomb
                    default: hit_count <= hit_count;
                endcase
            end
        end
    end

    // zapper logic
    logic zap_read, trig_read_noisy;
    assign zap_read = jc[0];
    assign trig_read_noisy = jc[1];

endmodule

`default_nettype wire

`default_nettype none

module game_fsm (
    input wire clk_65mhz,
    input wire reset_in,

    input wire [10:0] hcount_in, // horizontal index of current
pixel (0..1023)
    input wire [9:0]  vcount_in, // vertical index of current
pixel (0..767)
```

```
    input wire trigger_in,
    input wire active_in,
    input wire hit_valid,
    input wire [11:0] flash_in, hit_target,

    input wire left_button_in,
    input wire right_button_in,
    input wire down_button_in,

    output logic [11:0] pixel_out,

    output logic [3:0] difficulty,

    output logic [11:0] targets_onscreen,
    output logic [11:0] timer_out,
    output logic new_game_out
    );

    localparam  SCREEN_HEIGHT       = 768;
    localparam  SCREEN_WIDTH        = 1024;
    localparam  TARGET_WIDTH        = 64;
    localparam  CYCLES_PER_FRAME    = 1083264;
    localparam  GAME_TIME           = 3600; // 3600 frames aka
one minute

    localparam  CHERRY = 3'b000,
```

```
                STRAWBERRY = 3'b001,
                ORANGE = 3'b011,
                APPLE = 3'b111,
                BOMB = 3'b110,
                BACKGROUND = 3'b100;

    logic [20:0] cycle_counter;
    logic [11:0] frame_counter;

    enum {MAINMENU, SCORES, SELECTMODE, PLAYING, POPUP,
SAVESCREEN} state;

    logic target_hit [11:0];
    always_comb begin
        for (int i = 0; i < 12; i++)
            target_hit[i] = hit_target[i] && hit_valid ? 1'b1 :
1'b0;
    end

    logic game_target_drawing [4:0];
    logic game_target_activated [4:0];
    logic game_targets_prev_onscreen [4:0];
    logic [3:0] onscreen_sum;
    always_comb begin
        onscreen_sum = 0;
        for (int i = 0; i < 5; i++) begin
            onscreen_sum += game_target_activated[i];
```

```
        end
    end
    logic [4:0] game_targets_onscreen;
    logic [9:0] game_target_x [4:0];
    logic [9:0] game_target_y [4:0];
    logic [15:0] spawn_rng_num;
    logic [4:0] spawn_rng_reg;
    RNG #(.INITIAL_STATE(16'd9494))
spawn_rng(.clk_65mhz(clk_65mhz), .reset_in(reset_in),
                                     .trigger_next_state_in(1'b1
), .rng_out(spawn_rng_num));


    logic popup_gameover;
    logic play_mode;

    game_target #(.RNG_SEED(16'd123))

game_target_0(.clk_65mhz(clk_65mhz), .reset_in(reset_in),
                    .hcount_in(hcount_in), .vcount_in(vcount_in)
, .difficulty_in(difficulty),
                    .active_in(active_in &&
game_target_activated[0]), .is_hit(target_hit[7]),
                    .is_onscreen(game_targets_onscreen[0]), .is_
drawing(game_target_drawing[0]),
                    .target_x(game_target_x[0]), .target_y(game_
target_y[0]));
```

```
    game_target #(.RNG_SEED(16'd456))

game_target_1(.clk_65mhz(clk_65mhz), .reset_in(reset_in),
                    .hcount_in(hcount_in), .vcount_in(vcount_in)
, .difficulty_in(difficulty),
                    .active_in(active_in &&
game_target_activated[1]), .is_hit(target_hit[8]),
                    .is_onscreen(game_targets_onscreen[1]), .is_
drawing(game_target_drawing[1]),
                    .target_x(game_target_x[1]), .target_y(game_
target_y[1]));

    game_target #(.RNG_SEED(16'd6294))

game_target_2(.clk_65mhz(clk_65mhz), .reset_in(reset_in),
                    .hcount_in(hcount_in), .vcount_in(vcount_in)
, .difficulty_in(difficulty),
                    .active_in(active_in &&
game_target_activated[2]), .is_hit(target_hit[9]),
                    .is_onscreen(game_targets_onscreen[2]), .is_
drawing(game_target_drawing[2]),
                    .target_x(game_target_x[2]), .target_y(game_
target_y[2]));

    game_target #(.RNG_SEED(16'd794))
```

```
game_target_3(.clk_65mhz(clk_65mhz), .reset_in(reset_in),
                    .hcount_in(hcount_in), .vcount_in(vcount_in)
, .difficulty_in(difficulty),
                    .active_in(active_in &&
game_target_activated[3]), .is_hit(target_hit[10]),
                    .is_onscreen(game_targets_onscreen[3]), .is_
drawing(game_target_drawing[3]),
                    .target_x(game_target_x[3]), .target_y(game_
target_y[3]));

   popup_target #(.RNG_SEED(16'd18447))

popup_target_1(.clk_65mhz(clk_65mhz), .reset_in(reset_in),
                    .hcount_in(hcount_in), .vcount_in(vcount_in)
, .difficulty_in(difficulty),
                    .active_in(active_in &&
game_target_activated[4]), .is_hit(target_hit[11]),
                    .is_onscreen(game_targets_onscreen[4]), .is_
drawing(game_target_drawing[4]),
                    .target_x(game_target_x[4]), .target_y(game_
target_y[4]));

    logic [2:0] in_type;
    logic [10:0] rom_x, rom_width;
    logic [9:0] rom_y, rom_height;
    logic [11:0] rom_pixel;
```

```
    sprite_rom
rom(.pixel_clk_in(clk_65mhz), .sprite_type(in_type), .x_in(rom_x
), .y_in(rom_y),
                    .hcount_in(hcount_in-
3), .vcount_in(vcount_in), .width(rom_width), .height(rom_height
),
                    .pixel_out(rom_pixel));

    always_comb begin
        if(game_target_drawing[4]) begin
            in_type = APPLE;
            rom_x = game_target_x[4];
            rom_y = game_target_y[4];
            rom_width = 128;
            rom_height = 128;
        end else if(game_target_drawing[3]) begin
            in_type = ORANGE;
            rom_x = game_target_x[3];
            rom_y = game_target_y[3];
            rom_width = 128;
            rom_height = 128;
        end else if(game_target_drawing[2]) begin
            in_type = BOMB;
            rom_x = game_target_x[2];
            rom_y = game_target_y[2];
            rom_width = 128;
            rom_height = 128;
```

```
    end else if(game_target_drawing[1]) begin
        in_type = STRAWBERRY;
        rom_x = game_target_x[1];
        rom_y = game_target_y[1];
        rom_width = 128;
        rom_height = 128;
    end else if(game_target_drawing[0]) begin
        in_type = CHERRY;
        rom_x = game_target_x[0];
        rom_y = game_target_y[0];
        rom_width = 128;
        rom_height = 128;
    end else begin
        in_type = 0;
        rom_x = 0;
        rom_y = 0;
        rom_width = 0;
        rom_height = 0;
    end
end

logic [10:0] menu_target_x [6:0];
logic [9:0] menu_target_y [6:0];
logic [10:0] menu_target_w [6:0];
logic [9:0] menu_target_h [6:0];
logic [6:0] menu_targets_onscreen;
logic menu_target_drawing [6:0];
```

```
    menu_button menu_zero
( .x_in(menu_target_x[0]), .width(menu_target_w[0]), .y_in(menu_
target_y[0]), .height(menu_target_h[0]),
                             .hcount_in(hcount_in), .vcount_in(v
count_in), .drawing(menu_target_drawing[0]));

    menu_button menu_one
( .x_in(menu_target_x[1]), .width(menu_target_w[1]), .y_in(menu_
target_y[1]), .height(menu_target_h[1]),
                             .hcount_in(hcount_in), .vcount_in(v
count_in), .drawing(menu_target_drawing[1]));

    menu_button menu_two
( .x_in(menu_target_x[2]), .width(menu_target_w[2]), .y_in(menu_
target_y[2]), .height(menu_target_h[2]),
                             .hcount_in(hcount_in), .vcount_in(v
count_in), .drawing(menu_target_drawing[2]));

    menu_button
menu_three( .x_in(menu_target_x[3]), .width(menu_target_w[3]), .
y_in(menu_target_y[3]), .height(menu_target_h[3]),
                             .hcount_in(hcount_in), .vcount_in(v
count_in), .drawing(menu_target_drawing[3]));
```

```
    menu_button menu_four
( .x_in(menu_target_x[4]), .width(menu_target_w[4]), .y_in(menu_
target_y[4]), .height(menu_target_h[4]),
                             .hcount_in(hcount_in), .vcount_in(v
count_in), .drawing(menu_target_drawing[4]));

    menu_button menu_five
( .x_in(menu_target_x[5]), .width(menu_target_w[5]), .y_in(menu_
target_y[5]), .height(menu_target_h[5]),
                             .hcount_in(hcount_in), .vcount_in(v
count_in), .drawing(menu_target_drawing[5]));

    menu_button menu_six
( .x_in(menu_target_x[6]), .width(menu_target_w[6]), .y_in(menu_
target_y[6]), .height(menu_target_h[6]),
                             .hcount_in(hcount_in), .vcount_in(v
count_in), .drawing(menu_target_drawing[6]));

    always_comb begin
        case (state)
            MAINMENU: begin
                menu_targets_onscreen = 7'b0000011;
                for (int i = 0; i < 7; i++) begin
                    menu_target_x[i] = i == 0  || i == 1 ? 237 :
0;
                    menu_target_y[i] = i == 0 ? 30 : (i == 1 ?
400 : 0);
```

```
                    menu_target_w[i] = i == 0 || i == 1 ? 550 :
0;
                    menu_target_h[i] = i == 0 || i == 1 ? 350 :
0;
                end
            end

            SCORES : begin
                menu_targets_onscreen = 7'b0000001;
                for (int i = 0; i < 7; i++) begin
                    menu_target_x[i] = i == 0 ? 237 : 0;
                    menu_target_y[i] = i == 0 ? 400 : 0;
                    menu_target_w[i] = i == 0 ? 550 : 0;
                    menu_target_h[i] = i == 0 ? 350 : 0;
                end
            end

            SELECTMODE : begin
                menu_targets_onscreen = 7'b0011111;
                menu_target_x[0] = 237;
                menu_target_y[0] = 30;
                menu_target_w[0] = 260;
                menu_target_h[0] = 216;

                menu_target_x[1] = 527;
                menu_target_y[1] = 30;
                menu_target_w[1] = 260;
```

```
    menu_target_h[1] = 216;

    menu_target_x[2] = 237;
    menu_target_y[2] = 276;
    menu_target_w[2] = 260;
    menu_target_h[2] = 216;

    menu_target_x[3] = 527;
    menu_target_y[3] = 276;
    menu_target_w[3] = 260;
    menu_target_h[3] = 216;

    menu_target_x[4] = 237;
    menu_target_y[4] = 522;
    menu_target_w[4] = 550;
    menu_target_h[4] = 216;

    for (int i = 5; i < 7; i++) begin
        menu_target_x[i] = 0;
        menu_target_y[i] = 0;
        menu_target_w[i] = 0;
        menu_target_h[i] = 0;
    end
end

SAVESCREEN : begin
    menu_targets_onscreen = 7'b1111111;
```

```
menu_target_x[0] = 0;
menu_target_y[0] = 0;
menu_target_w[0] = 0;
menu_target_h[0] = 0;

menu_target_x[1] = 0;
menu_target_y[1] = 0;
menu_target_w[1] = 0;
menu_target_h[1] = 0;

menu_target_x[2] = 0;
menu_target_y[2] = 0;
menu_target_w[2] = 0;
menu_target_h[2] = 0;

menu_target_x[3] = 0;
menu_target_y[3] = 0;
menu_target_w[3] = 0;
menu_target_h[3] = 0;

menu_target_x[4] = 0;
menu_target_y[4] = 0;
menu_target_w[4] = 0;
menu_target_h[4] = 0;

menu_target_x[5] = 0;
menu_target_y[5] = 0;
```

```
                menu_target_w[5] = 0;
                menu_target_h[5] = 0;

                menu_target_x[6] = 0;
                menu_target_y[6] = 0;
                menu_target_w[6] = 0;
                menu_target_h[6] = 0;
            end

            default : begin
                menu_targets_onscreen = 7'b0000000;
                for (int i = 0; i < 7; i++) begin
                    menu_target_x[i] = 0;
                    menu_target_y[i] = 0;
                    menu_target_w[i] = 0;
                    menu_target_h[i] = 0;
                end
            end
        endcase
    end

    assign targets_onscreen = {state == PLAYING || state ==
POPUP ? game_targets_onscreen : 5'h0, state != PLAYING ?
menu_targets_onscreen : 7'h0};

    logic next_frame_blank, blank_frame;
```

```
    always_ff @ (posedge clk_65mhz) begin
        if (reset_in) begin
            state <= MAINMENU;
            difficulty <= 0;
            cycle_counter <= 0;
            frame_counter <= 0;
            next_frame_blank <= 0;
            blank_frame <= 0;
            spawn_rng_reg <= 0;
            game_target_activated[0] <= 1;
            game_target_activated[1] <= 1;
            game_target_activated[2] <= 0;
            game_target_activated[3] <= 0;
            game_target_activated[4] <= 0;
            play_mode <= 0;
        end else begin
            spawn_rng_reg <= spawn_rng_num[4:0];
            if (trigger_in) begin
                next_frame_blank <= 1;
                blank_frame <= 0;
            end else if (next_frame_blank == 1 && hcount_in ==
1343 && vcount_in == 805) begin
                next_frame_blank <= 0;
                blank_frame <= 1;
            end else if (blank_frame && hcount_in == 1343 &&
vcount_in == 805 && !(|flash_in)) begin
                next_frame_blank <= 0;
```

```
                blank_frame <= 0;
            end
            case (state)
                MAINMENU : begin
                    if(menu_target_drawing[0]) begin
                        if(flash_in[0]) pixel_out <= 12'hFFF;
                        else if(blank_frame) pixel_out <= 12'h1;
                        else pixel_out <= 12'h1;
                    end else if(menu_target_drawing[1]) begin
                        if(flash_in[1]) pixel_out <= 12'hFFF;
                        else if(blank_frame) pixel_out <= 12'h1;
                        else pixel_out <= 12'h1;
                    end else
                        pixel_out <= 12'h0;
                    if (target_hit[0]) begin
                        state <= SELECTMODE;
                    end else if (target_hit[1]) begin
                        state <= SCORES;
                    end
                end

                SCORES : begin
                    if(menu_target_drawing[0] && flash_in[0])
pixel_out <= 12'hFFF;
                    else pixel_out <= blank_frame &&
menu_target_drawing[0] ? 12'h1 : (menu_target_drawing[0] ?
12'h800 : 12'h0);
```

```
                if (target_hit[0]) begin
                    state <= MAINMENU;
                end
            end

            SELECTMODE : begin
                if(menu_target_drawing[0]) begin
                    if(flash_in[0]) pixel_out <= 12'hFFF;
                    else if(blank_frame) pixel_out <= 12'h1;
                    else pixel_out <= difficulty == 0 ?
12'h0F0 : 12'h1;
                end else if (menu_target_drawing[1]) begin
                    if(flash_in[1]) pixel_out <= 12'hFFF;
                    else if(blank_frame) pixel_out <= 12'h1;
                    else pixel_out <= difficulty == 1 ?
12'h0F0 : 12'h1;
                end else if (menu_target_drawing[2]) begin
                    if(flash_in[2]) pixel_out <= 12'hFFF;
                    else if(blank_frame) pixel_out <= 12'h1;
                    else pixel_out <= play_mode == 0 ?
12'h00F : 12'h1;
                end else if (menu_target_drawing[3]) begin
                    if(flash_in[3]) pixel_out <= 12'hFFF;
                    else if(blank_frame) pixel_out <= 12'h1;
                    else pixel_out <= play_mode == 1 ?
12'h00F : 12'h1;
                end else if (menu_target_drawing[4]) begin
```

```
                if(flash_in[4]) pixel_out <= 12'hFFF;
                else if(blank_frame) pixel_out <= 12'h1;
                else pixel_out <= 12'h1;
            end else
                pixel_out <= 12'h0;
            if (target_hit[0]) begin
                // difficulty <= (difficulty == 0) ? 0 :
difficulty - 1;
                difficulty <= 4'b0;
            end else if (target_hit[1]) begin
                // difficulty <= (difficulty == 7) ? 7 :
difficulty + 1;
                difficulty <= 4'b1;
            end else if (target_hit[2]) begin
                play_mode <= 0;
            end else if (target_hit[3]) begin
                play_mode <= 1;
            end else if (target_hit[4] ||
down_button_in) begin
                state <= play_mode ? POPUP : PLAYING;
                new_game_out <= 1'b1;
                timer_out <= 60;
            end
        end

        PLAYING : begin
            new_game_out <= 1'b0;
```

```
                    if(game_target_drawing[4]) begin
                        if(flash_in[11]) pixel_out <=
|rom_pixel ? 12'hFFF : 0;
                        else if (blank_frame) pixel_out <=
12'h1;
                        else pixel_out <= rom_pixel;
                    end else if (game_target_drawing[3]) begin
                        if(flash_in[10]) pixel_out <=
|rom_pixel ? 12'hFFF : 0;
                        else if (blank_frame) pixel_out <=
12'h1;
                        else pixel_out <= rom_pixel;
                    end else if (game_target_drawing[2]) begin
                        if(flash_in[9]) pixel_out <=
|rom_pixel ? 12'hFFF : 0;
                        else if (blank_frame) pixel_out <=
12'h1;
                        else pixel_out <= rom_pixel;
                    end else if (game_target_drawing[1]) begin
                        if(flash_in[8]) pixel_out <=
|rom_pixel ? 12'hFFF : 0;
                        else if (blank_frame) pixel_out <=
12'h1;
                        else pixel_out <= rom_pixel;
                    end else if (game_target_drawing[0]) begin
                        if(flash_in[7]) pixel_out <=
|rom_pixel ? 12'hFFF : 0;
```

```
                    else if (blank_frame) pixel_out <=
12'h1;
                    else pixel_out <= rom_pixel;
                end else
                    pixel_out <= blank_frame ? 12'h1 :
12'h0;
                if (frame_counter == (GAME_TIME / 3)) begin
                    difficulty <= difficulty + 1;
                end else if (frame_counter == (2 * GAME_TIME
/ 3)) begin
                    difficulty <= difficulty + 1;
                end

                for(int i = 0; i < 5; i++) begin
                    game_targets_prev_onscreen[i] <=
game_targets_onscreen[i];
                end
                if(onscreen_sum < 2) begin
                    if(spawn_rng_reg < 12
&& !game_target_activated[0]) game_target_activated[0] <= 1'b1;
                    else if (spawn_rng_reg < 20
&& !game_target_activated[1]) game_target_activated[1] <= 1'b1;
                    else if (spawn_rng_reg < 26
&& !game_target_activated[2]) game_target_activated[2] <= 1'b1;
                    else if (spawn_rng_reg < 29
&& !game_target_activated[3]) game_target_activated[3] <= 1'b1;
```

```
                    else if (!game_target_activated[4])
game_target_activated[4] <= 1'b1;
                end else begin
                    for(int i = 0; i < 5; i++) begin
                        if(game_targets_prev_onscreen[i] !=
game_targets_onscreen[i] && game_targets_onscreen[i] == 0)
                            game_target_activated[i] = 1'b0;
                    end
                end

                if (frame_counter == GAME_TIME) begin // one
minute has passed, game over
                    state <= MAINMENU;
                    cycle_counter <= 0;
                    frame_counter <= 0;
                    difficulty <= 0;
                    timer_out <= 0;
                end else if(active_in) begin
                    if (cycle_counter == CYCLES_PER_FRAME)
begin
                        cycle_counter <= 0;
                        frame_counter <= frame_counter + 1;
                    end else begin
                        cycle_counter <= cycle_counter + 1;
                    end

                    if (frame_counter % 60 == 0) begin
```

```
                    timer_out     <= timer_out - 1;
                end
            end
        end

        POPUP : begin
            new_game_out <= 1'b0;
            game_target_activated[4] <= 1'b1;
            if(game_target_drawing[4]) begin
                if(flash_in[11]) pixel_out <=
|rom_pixel ? 12'hFFF : 0;
                else if(blank_frame) pixel_out <= 12'h1;
                else pixel_out <= rom_pixel;
            end else pixel_out <= blank_frame ? 12'h1 :
12'h0;

            // if (popup_gameover) begin
            //     state <= MAINMENU;
            // end


            if (frame_counter == GAME_TIME) begin // one
minute has passed, game over
                state <= MAINMENU;
                cycle_counter <= 0;
                frame_counter <= 0;
                difficulty <= 0;
```

```
                    timer_out <= 0;
                end else if(active_in) begin
                    if (cycle_counter == CYCLES_PER_FRAME)
begin
                        cycle_counter <= 0;
                        frame_counter <= frame_counter + 1;
                    end else begin
                        cycle_counter <= cycle_counter + 1;
                    end

                    if (frame_counter % 60 == 0) begin
                        timer_out     <= timer_out - 1;
                    end
                end
            end

            SAVESCREEN : begin
            end

        endcase
    end
end


endmodule // menu_fsm

`default_nettype wire
```

```
////////////////////////////////////////////////////////////////
//////
//
// blob: generate rectangle on screen
//
////////////////////////////////////////////////////////////////
//////
module menu_button
   (input wire [10:0] x_in, hcount_in, width,
    input wire [9:0] y_in, vcount_in, height,
    output logic drawing);

   always_comb begin
      if  ((hcount_in >= x_in && hcount_in < (x_in+width)) &&
            (vcount_in >= y_in && vcount_in < (y_in+height)))
        drawing = 1'b1;
      else
        drawing = 1'b0;
   end
endmodule

module hit_detect(
    input wire clk,
    input wire reset,
```

```
    input wire zap_in,
    input wire trig_in,
    input wire [11:0] targets_onscreen,
    input wire [10:0] hcount_in,
    input wire [9:0] vcount_in,
    output logic valid_out, targets_update,
    output logic [11:0] flash_out,
    output logic [11:0] target_hit
);

localparam  IDLE            = 3'b000,
            TRIGGERED       = 3'b001,
            TARGET_CHECK    = 3'b011,
            HIT             = 3'b111,
            MISS            = 3'b110,
            ASSERT          = 3'b100;

localparam CYCLES_PER_FRAME = 1083264;

logic blank_zap, next_frame, trigger_pressed;
logic [2:0] current_state, next_state;
logic [2:0] entries;
logic [3:0] flash_count, index_counter;
logic [3:0] index_to_flash [6:0];
logic [11:0] target_buffer, next_flash;
logic [20:0] wait_counter;
```

```
always_comb begin
    next_frame = hcount_in == 1343 && vcount_in == 805;
    next_flash = 12'h0;
    next_flash[index_to_flash[flash_count]] = 1'b1;
    case(current_state)
        IDLE        : next_state = trigger_pressed && next_frame
&& targets_onscreen ? TRIGGERED : IDLE;
        TRIGGERED   : next_state = next_frame ? TARGET_CHECK :
TRIGGERED;
        TARGET_CHECK: next_state = blank_zap != zap_in &&
blank_zap == 1 && next_frame ? HIT : next_frame ? MISS :
TARGET_CHECK;
        HIT         : next_state = ASSERT;
        MISS        : next_state = flash_count < entries ?
TARGET_CHECK : ASSERT;
        ASSERT      : next_state = IDLE;
        default     : next_state = IDLE;
    endcase
end

always_ff @(posedge clk) begin
    if(reset) begin
        blank_zap <= 1'b0;
        trigger_pressed <= 1'b0;
        current_state <= IDLE;
        entries <= 3'h0;
        index_counter <= 4'h0;
```

```
        flash_count <= 4'h0;
        for (int i = 0; i < 7; i++) index_to_flash[i] <= 0;
        target_buffer <= 12'h0;
        wait_counter <= 21'h0;
        valid_out <= 1'b0;
        targets_update <= 1'b0;
        flash_out <= 12'h0;
        target_hit <= 12'h0;
    end else begin
        current_state <= next_state;
        case(current_state)
            IDLE :  begin
                            targets_update <= 1'b1;
                            flash_out <= 12'h0;
                            if(trig_in == 1) trigger_pressed <= 1;
                            if(next_state == TRIGGERED) begin
                                wait_counter <= CYCLES_PER_FRAME;
                                target_buffer <= targets_onscreen;
                                entries <= 3'h0;
                                index_counter <= 4'h0;
                                flash_count <= 4'h0;
                                for (int i = 0; i < 7; i++)
index_to_flash[i] <= 0;
                            end
                        end

            TRIGGERED : begin
```

```
                    if(next_state == TARGET_CHECK) begin
                        wait_counter <= CYCLES_PER_FRAME;
                        flash_out <= next_flash;
                    end else begin
                        if(|target_buffer) begin
                            target_buffer <= target_buffer
>> 1;
                            index_counter <= index_counter +
1;
                            if(target_buffer[0]) begin
                                index_to_flash[entries] <=
index_counter;
                                entries <= entries + 1;
                            end
                        end
                        wait_counter <= wait_counter - 1;
                        blank_zap <= zap_in;
                    end
                end

            TARGET_CHECK : begin
                    if(next_state == MISS) flash_count <=
flash_count + 1;
                    wait_counter <= wait_counter - 1;
                end

            HIT : begin
```

```
                        valid_out <= 1'b1;
                        target_hit[index_to_flash[flash_count]]
<= 1'b1;
                  end

            MISS : begin
                   if(next_state == ASSERT) begin
                       valid_out <= 1'b1;
                       target_hit <= 12'h0;
                   end else if(next_state == TARGET_CHECK)
begin
                       flash_out <= next_flash;
                       wait_counter <= CYCLES_PER_FRAME;
                   end
                  end

            ASSERT : begin
                       valid_out <= 0;
                       target_hit <= 0;
                       trigger_pressed <= 0;
                   end
        endcase
    end
end
endmodule
`default_nettype none
```

```
module game_target
    #(parameter   RNG_SEED = 0)
    (
    input wire clk_65mhz,
    input wire reset_in,

    input wire active_in,
    input wire [3:0] difficulty_in,
    input wire is_hit,

    input wire [10:0] hcount_in, // horizontal index of current
pixel (0..1023)
    input wire [9:0]  vcount_in, // vertical index of current
pixel (0..767)

    output logic is_onscreen, is_drawing,
    output logic [9:0] target_x, target_y
    );

    localparam  SCREEN_HEIGHT       = 768;
    localparam  SCREEN_WIDTH        = 1024;
    localparam  TARGET_WIDTH        = 128;
    localparam  TERMINAL_VELOCITY   = 24;

    enum {VISIBLE, MISSED, NONVISIBLE} state;
    logic [1:0]     GRAVITY;
    logic [7:0]     INITIAL_VELOCITY;
```

```
    assign GRAVITY           = (difficulty_in > 1'b0) ? 2 : 1;
    assign INITIAL_VELOCITY = (difficulty_in > 1'b0) ? 40: 20;

    logic [9:0] vertical_velocity;
    logic [9:0] horizontal_velocity;

    logic [15:0] rng_val;
    logic trigger_next_rng_state;

    RNG #(.INITIAL_STATE(RNG_SEED))
my_rng( .clk_65mhz(clk_65mhz), .reset_in(reset_in),
                                            .trigger_next_state_
in(trigger_next_rng_state),
                                            .rng_out(rng_val));

    // enum {UP, DOWN} vertical_direction;
    // enum {LEFT, RIGHT} horizontal_direction;

    logic moving_up;
    logic moving_right;
        logic hit_once;

    logic [4:0] frames_to_wait; // wait up to 32 frames or
roughly a half second

    assign is_onscreen = state == VISIBLE ? 1'b1 : 1'b0;
```

```
    always_ff @ (posedge clk_65mhz) begin
        if (reset_in) begin
            target_y                                <=
SCREEN_HEIGHT;
            trigger_next_rng_state  <= 1'b1; // TODO: is this
needed?
            vertical_velocity           <= INITIAL_VELOCITY +
(rng_val % 10);
            horizontal_velocity         <= 3 + (rng_val % 9);
            target_x                                <= rng_val %
SCREEN_WIDTH;

            moving_up                           <= 1'b1;
            moving_right                        <= ((rng_val %
SCREEN_WIDTH) < (SCREEN_WIDTH / 2)) ? 1'b1 : 1'b0; // if on
right side of screen, target starts moving left, and vice versa
                        hit_once
<= 1'b0;

            state <= VISIBLE;

        end else if ((hcount_in == 0) && (vcount_in == 0) &&
active_in) begin
            case (state)
                VISIBLE : begin
                    trigger_next_rng_state <= 1'b0;  // TODO: is
this needed?
```

```
                    target_y <= moving_up ? target_y -
vertical_velocity : target_y + vertical_velocity;
                    target_x <= moving_right ? target_x +
horizontal_velocity : target_x - horizontal_velocity;
                    // this case block determines velocity
changes
                    case (moving_up)
                        1'b1: begin // moving up
                            if (vertical_velocity < GRAVITY)
begin
                                vertical_velocity <= 0;
                                moving_up <= 1'b0; // reached
apex of trajectory, begin moving down
                            end else begin // still moving
upwards
                                vertical_velocity <=
vertical_velocity - GRAVITY;
                            end
                        end
                        1'b0: begin // moving down
                            if ((vertical_velocity + GRAVITY) >
TERMINAL_VELOCITY) begin
                                vertical_velocity <=
TERMINAL_VELOCITY;
                            end else begin
                                vertical_velocity <=
vertical_velocity + GRAVITY;
```

```
                    end
                end
            endcase

            // check if off screen
            if (!moving_up && ((target_y +
vertical_velocity) > SCREEN_HEIGHT)) begin
                state <= MISSED;
            end else if ((vertical_velocity > target_y)
&& moving_up) begin
                moving_up <= 1'b0;
            end
            if (is_hit) begin
                state <= NONVISIBLE;
                trigger_next_rng_state <= 1'b1;
            end
        end


        MISSED : begin
            // this state is primarily for scoring
purposes
            // we spend exactly one frame here
            trigger_next_rng_state <= 1'b1;
            state <= NONVISIBLE;
        end
```

```
                NONVISIBLE : begin
                    if (trigger_next_rng_state == 1'b1) begin //
just entered state
                        trigger_next_rng_state <= 1'b0;
                        frames_to_wait <= (rng_val % 30);
                    end else if (frames_to_wait == 0) begin
                        target_y <= SCREEN_HEIGHT;
                        // trigger_next_rng_state <= 1'b1;
                        vertical_velocity <= INITIAL_VELOCITY +
(rng_val % 10);
                        horizontal_velocity <= 3 + (rng_val %
9);
                        target_x = rng_val % SCREEN_WIDTH;

                        moving_up = 1'b1;
                        moving_right =  ((rng_val %
SCREEN_WIDTH) < (SCREEN_WIDTH / 2)) ? 1'b1 : 1'b0; // if on
right side of screen, target starts moving left, and vice versa

                        state <= VISIBLE;
                    end else begin
                        frames_to_wait <= frames_to_wait - 1;
                    end

                end
            endcase
        end else if (is_hit) begin
```

```
                if (difficulty_in == 3 && !hit_once) begin
                    moving_right <= !moving_right;
                    moving_up <= 1'b1;
                    vertical_velocity <= vertical_velocity +
(INITIAL_VELOCITY / 3);
                end else begin
                    state <= NONVISIBLE;
                    trigger_next_rng_state <= 1'b1;
                    hit_once <= 1'b0;
               end
           end
       end // end always_ff

       always_comb begin
           is_onscreen = state == VISIBLE ? 1'b1 : 1'b0;
           if( hcount_in >= target_x && hcount_in < (target_x +
TARGET_WIDTH) &&
               vcount_in >= target_y && vcount_in < (target_y +
TARGET_WIDTH)) begin
                   if(state == VISIBLE /*&& active_in*/)
                       is_drawing = 1'b1;
                   else
                       is_drawing = 1'b0;
           end else
               is_drawing = 1'b0;
       end
```

```
endmodule

`default_nettype wire
`default_nettype none

module popup_target
    #(parameter   RNG_SEED = 0)
    (
    input wire clk_65mhz,
    input wire reset_in,

    input wire active_in,
    input wire [3:0] difficulty_in,
    input wire is_hit,

    input wire [10:0] hcount_in, // horizontal index of current
pixel (0..1023)
    input wire [9:0]  vcount_in, // vertical index of current
pixel (0..767)

    output logic is_onscreen, is_drawing,
    output logic [9:0] target_x, target_y,
        output logic gameover
    );

    localparam  SCREEN_HEIGHT       = 768;
    localparam  SCREEN_WIDTH        = 1024;
```

```
    localparam  TARGET_WIDTH         = 128;
    localparam  TERMINAL_VELOCITY    = 24;

    enum {VISIBLE, MISSED, NONVISIBLE} state;
    logic [1:0]     GRAVITY;
    logic [7:0]     INITIAL_VELOCITY;
    assign GRAVITY          = (difficulty_in > 1'b0) ? 2 : 1;
    assign INITIAL_VELOCITY = (difficulty_in > 1'b0) ? 35: 25;

    logic [9:0] vertical_velocity;
    logic [9:0] horizontal_velocity;
        logic [31:0] hit_count;

    logic [15:0] rng_val;
    logic trigger_next_rng_state;

    RNG #(.INITIAL_STATE(RNG_SEED))
my_rng( .clk_65mhz(clk_65mhz), .reset_in(reset_in),
                                            .trigger_next_state_
in(trigger_next_rng_state),
                                            .rng_out(rng_val));

    // enum {UP, DOWN} vertical_direction;
    // enum {LEFT, RIGHT} horizontal_direction;

    logic moving_up;
    logic moving_right;
```

```
    logic [4:0] frames_to_wait; // wait up to 32 frames or
roughly a half second

    assign is_onscreen = state == VISIBLE ? 1'b1 : 1'b0;

    always_ff @ (posedge clk_65mhz) begin
        if (reset_in) begin
            target_y                               <=
SCREEN_HEIGHT;
            trigger_next_rng_state  <= 1'b1; // TODO: is this
needed?
            vertical_velocity   <= INITIAL_VELOCITY + (rng_val %
10);
            horizontal_velocity <= 3 + (rng_val % 9);
            target_x    <= rng_val % SCREEN_WIDTH;
            moving_up   <= 1'b1;
            moving_right    <= ((rng_val % SCREEN_WIDTH) <
(SCREEN_WIDTH / 2)) ? 1'b1 : 1'b0; // if on right side of
screen, target starts moving left, and vice versa
            hit_count   <= 32'b0;
            state <= VISIBLE;

        end else if ((hcount_in == 0) && (vcount_in == 0) &&
active_in) begin
            case (state)
                VISIBLE : begin
```

```
                    trigger_next_rng_state <= 1'b0;  // TODO: is
this needed?
                    target_y <= moving_up ? target_y -
vertical_velocity : target_y + vertical_velocity;
                    target_x <= moving_right ? target_x +
horizontal_velocity : target_x - horizontal_velocity;
                    // this case block determines velocity
changes
                    case (moving_up)
                        1'b1: begin // moving up
                            if (vertical_velocity < GRAVITY)
begin
                                vertical_velocity <= 0;
                                moving_up <= 1'b0; // reached
apex of trajectory, begin moving down
                            end else begin // still moving
upwards
                                vertical_velocity <=
vertical_velocity - GRAVITY;
                            end
                        end
                        1'b0: begin // moving down
                            if ((vertical_velocity + GRAVITY) >
TERMINAL_VELOCITY) begin
                                vertical_velocity <=
TERMINAL_VELOCITY;
                            end else begin
```

```
                                vertical_velocity <=
vertical_velocity + GRAVITY;
                            end
                        end
                    endcase

                    // check if off screen
                    if (!moving_up && ((target_y +
vertical_velocity) > SCREEN_HEIGHT)) begin
                        state <= MISSED;
                    end else if ((vertical_velocity > target_y)
&& moving_up) begin
                        moving_up <= 1'b0;
                    end
                    if (is_hit) begin
                        state <= NONVISIBLE;
                        trigger_next_rng_state <= 1'b1;
                    end

                end
                MISSED : begin
                                    // target missed, game over!
                    state <= MISSED;
                                        gameover <=
difficulty_in == 3 ? 1 : 0;      // if difficulty is 3, end game
on
```

```
                                        trigger_next_rng_state
<= 1'b1;
                                        state <= NONVISIBLE;
                end

                                NONVISIBLE : begin
                                        if
(trigger_next_rng_state == 1'b1) begin // just entered state

trigger_next_rng_state <= 1'b0;
                                                frames_to_wait
<= (rng_val % 30);
                                        end else if
(frames_to_wait == 0) begin
                                                target_y <=
SCREEN_HEIGHT;
                                                //
trigger_next_rng_state <= 1'b1;

vertical_velocity <= INITIAL_VELOCITY + (rng_val % 10);

horizontal_velocity <= 3 + (rng_val % 9);
                                                target_x =
rng_val % SCREEN_WIDTH;
                                                moving_up =
1'b1;
```

```
                                                    moving_right =
((rng_val % SCREEN_WIDTH) < (SCREEN_WIDTH / 2)) ? 1'b1 : 1'b0;
// if on right side of screen, target starts moving left, and
vice versa
                                                    state <=
VISIBLE;
                                            end else begin
                                                    frames_to_wait
<= frames_to_wait - 1;
                                            end

                                    end

                endcase
            end else if (is_hit) begin
                                trigger_next_rng_state <= 1'b1;
                                hit_count <= hit_count + 1;
                                moving_right <= !moving_right;
                                moving_up <= 1'b1;
                                vertical_velocity <=
vertical_velocity + (INITIAL_VELOCITY / 3);
                                horizontal_velocity <= difficulty_in
+ 3 + (rng_val % 9);
                        end

        end // end always_ff
```

```
    always_comb begin
        is_onscreen = state == VISIBLE ? 1'b1 : 1'b0;
        if( hcount_in >= target_x && hcount_in < (target_x +
TARGET_WIDTH) &&
            vcount_in >= target_y && vcount_in < (target_y +
TARGET_WIDTH)) begin
                if(state == VISIBLE /*&& active_in*/)
                    is_drawing = 1'b1;
                else
                    is_drawing = 1'b0;
        end else
            is_drawing = 1'b0;
    end

endmodule

`default_nettype wire
module sprite_rom
   (input wire pixel_clk_in,
    input wire [2:0] sprite_type,
    input wire [10:0] x_in,hcount_in, width,
    input wire [9:0] y_in,vcount_in, height,
    output logic [11:0] pixel_out);

    localparam  WIDTH = 128;

    localparam  CHERRY = 3'b000,
```

```
                STRAWBERRY = 3'b001,
                ORANGE = 3'b011,
                APPLE = 3'b111,
                BOMB = 3'b110;

    logic [16:0] image_addr;   // num of bits for 128*640 ROM
    logic [9:0] y_offset;
    logic [7:0] image_bits, cm_addr, red_mapped, green_mapped,
blue_mapped;
    logic [11:0] next_out;
    assign next_out = {red_mapped[7:4], green_mapped[7:4],
blue_mapped[7:4]};

    logic [6:0] col;
    logic [16:0] row;
    assign col = (hcount_in - x_in);
    assign row = (vcount_in - y_in + y_offset);
    // calculate rom address and read the location
    assign image_addr = col + (row * WIDTH);
    targets_rom
rom1(.clka(pixel_clk_in), .addra(image_addr), .douta(image_bits)
);

    // use color map to create 4 bits R, 4 bits G, 4 bits B
    // since the image is greyscale, just replicate the red
pixels
    // and not bother with the other two color maps.
```

```
    targets_red_cm rcm
(.clka(pixel_clk_in), .addra(cm_addr), .douta(red_mapped));
    targets_green_cm gcm
(.clka(pixel_clk_in), .addra(cm_addr), .douta(green_mapped));
    targets_blue_cm bcm
(.clka(pixel_clk_in), .addra(cm_addr), .douta(blue_mapped));

    always_comb begin
        case(sprite_type)
            CHERRY: begin
                        y_offset = 0;
                    end
            STRAWBERRY: begin
                        y_offset = 128;
                        end
            ORANGE: begin
                        y_offset = 256;
                    end
            APPLE:  begin
                        y_offset = 384;
                    end
            BOMB:   begin
                        y_offset = 512;
                    end
            default: begin
                        y_offset = 0;
                     end
```

```
        endcase
    end

    // note the one clock cycle delay in pixel!
    always_ff @ (posedge pixel_clk_in) begin
        cm_addr <= image_bits;
        if ((hcount_in >= x_in+3 && hcount_in < (x_in+width)) &&
           (vcount_in >= y_in && vcount_in < (y_in+height))
            && next_out != 12'hFFF)
            pixel_out <= {next_out[11:1], 1'b1}; //allows
'black' pixels to not get covered by background
        else pixel_out <= 0;
    end
endmodule
module background_rom(  input wire pixel_clk_in,
                        input wire [9:0] vcount_in,
                        input wire [10:0] hcount_in,
                        output logic [11:0] pixel_out);

    localparam WIDTH = 256;

    logic [16:0] image_addr, row;
    logic [7:0] image_bits, cm_addr, red_mapped, green_mapped,
blue_mapped, col;
    logic [11:0] next_out;
    assign next_out = {red_mapped[7:4] >> 1, green_mapped[7:4]
>> 1, blue_mapped[7:4] >> 1};
```

```
    assign col = hcount_in % 256;

    bg_im_rom
rom1(.clka(pixel_clk_in), .addra(image_addr), .douta(image_bits)
);

    bg_red_cm
rcm(.clka(pixel_clk_in), .addra(cm_addr), .douta(red_mapped));
    bg_green_cm
gcm(.clka(pixel_clk_in), .addra(cm_addr), .douta(green_mapped));
    bg_blue_cm
bcm(.clka(pixel_clk_in), .addra(cm_addr), .douta(blue_mapped));

    always_comb begin
        if(vcount_in < 400) begin
            row = vcount_in;
            image_addr = (col + row * WIDTH) % (128*30);
        end else begin
            row = vcount_in - 400;
            image_addr = col + row * WIDTH;
        end
    end

    always_ff @(posedge pixel_clk_in) begin
        cm_addr <= image_bits;
        pixel_out <= next_out;
```

```
    end
endmodule
`default_nettype none

module RNG (
    input wire clk_65mhz,
    input wire reset_in,
    input wire trigger_next_state_in,
    output logic [15:0] rng_out);

    parameter INITIAL_STATE = 0;

    logic [15:0] rng_state;
    logic [15:0] temp1;
    logic [15:0] temp2;
    logic after_initial_state = 1'b0;


    // debating whether or not "trigger_next_state_in" is even
necessary or if
    // we can just have this cycling every clock cycle and the
    // randomness comes from the unpredictability of when you
call
    // it, although having a trigger makes testing the sequence
eaiser

    always_ff @ (posedge clk_65mhz) begin
```

```
        if (reset_in) begin
            rng_state <= INITIAL_STATE;
            after_initial_state <= 1'b0;
        end else if (trigger_next_state_in) begin
            // cycles through all 65,535 values in a
deterministic
            // order before repeating cycle
            temp1 = after_initial_state ? (rng_state ^
(rng_state << 5)) : (INITIAL_STATE ^ (INITIAL_STATE << 5));
            temp2 = temp1 ^ (temp1 >> 7);
            rng_state = temp2 + 16'd28383;

            rng_out <= rng_state;
            after_initial_state <= 1'b1;
        end else begin
            rng_out <= rng_state;
        end
    end

endmodule // RNG

`default_nettype wire
`default_nettype none

module seven_seg_controller(input wire         clk_in,
                            input wire         rst_in,
                            input wire [31:0]  val_in,
```

```
                        output logic[6:0]   cat_out,
                        output logic[7:0]   an_out
    );

    logic[7:0]      segment_state;
    logic[31:0]     segment_counter;
    logic [3:0]     routed_vals;
    logic [6:0]     led_out;

    binary_to_seven_seg my_converter
( .bin_in(routed_vals), .hex_out(led_out));
    assign cat_out = ~led_out;
    assign an_out = ~segment_state;


    always_comb begin
        case(segment_state)
            8'b0000_0001:   routed_vals = val_in[3:0];
            8'b0000_0010:   routed_vals = val_in[7:4];
            8'b0000_0100:   routed_vals = val_in[11:8];
            8'b0000_1000:   routed_vals = val_in[15:12];
            8'b0001_0000:   routed_vals = val_in[19:16];
            8'b0010_0000:   routed_vals = val_in[23:20];
            8'b0100_0000:   routed_vals = val_in[27:24];
            8'b1000_0000:   routed_vals = val_in[31:28];
            default:        routed_vals = val_in[3:0];
        endcase
```

```
    end

    always_ff @(posedge clk_in)begin
        if (rst_in)begin
            segment_state <= 8'b0000_0001;
            segment_counter <= 32'b0;
        end else begin
            if (segment_counter == 32'd100_000)begin
                segment_counter <= 32'd0;
                segment_state <=
{segment_state[6:0],segment_state[7]};
            end else begin
                segment_counter <= segment_counter +1;
            end
        end
    end

endmodule //seven_seg_controller

`default_nettype wire
`default_nettype none //prevents system from inferring an
undeclared logic
module binary_to_seven_seg(
        input wire [3:0]       bin_in,  //declaring input
explicitly
        output logic [6:0]      hex_out);  //declaring output
explicitly
```

```
    always_comb begin
        case(bin_in)
            4'b0000 : hex_out = 7'b0111111;
            4'b0001 : hex_out = 7'b0000110;
            4'b0010 : hex_out = 7'b1011011;
            4'b0011 : hex_out = 7'b1001111;
            4'b0100 : hex_out = 7'b1100110;
            4'b0101 : hex_out = 7'b1101101;
            4'b0110 : hex_out = 7'b1111101;
            4'b0111 : hex_out = 7'b0000111;
            4'b1000 : hex_out = 7'b1111111;
            4'b1001 : hex_out = 7'b1100111;
            4'b1010 : hex_out = 7'b1110111;
            4'b1011 : hex_out = 7'b1111100;
            4'b1100 : hex_out = 7'b0111001;
            4'b1101 : hex_out = 7'b1011110;
            4'b1110 : hex_out = 7'b1111001;
            4'b1111 : hex_out = 7'b1110001;
        endcase
      end
endmodule //binary_to_hex
`default_nettype wire

`default_nettype none

module display(
```

```
    input wire clk_65mhz,
    input wire reset,
    input wire [11:0] pixel,
    output logic [3:0] vga_r, vga_g, vga_b,
    output logic [10:0] hcount,
    output logic [9:0] vcount,
    output logic hsync, vsync, blank,
    output logic vga_hsync, vga_vsync
    );

    xvga
xvga0(.vclock_in(clk_65mhz), .reset(reset), .hcount_out(hcount),
.vcount_out(vcount),
            .hsync_out(hsync), .vsync_out(vsync), .blank_out(bla
nk));

    always_comb begin
        vga_r = ~blank ? pixel[11:8] : 0;
        vga_g = ~blank ? pixel[7:4]  : 0;
        vga_b = ~blank ? pixel[3:0]  : 0;

        vga_hsync = ~hsync;
        vga_vsync = ~vsync;
    end

endmodule
```

```
`default_nettype wire
`default_nettype none

module xvga(input wire vclock_in,
            input wire reset,
            output logic [10:0] hcount_out,    // pixel number
on current line
            output logic [9:0] vcount_out,     // line number
            output logic vsync_out, hsync_out,
            output logic blank_out);

   parameter DISPLAY_WIDTH  = 1024;      // display width
   parameter DISPLAY_HEIGHT = 768;       // number of lines

   parameter  H_FP = 24;                 // horizontal front
porch
   parameter  H_SYNC_PULSE = 136;        // horizontal sync
   parameter  H_BP = 160;                // horizontal back
porch

   parameter  V_FP = 3;                  // vertical front porch
   parameter  V_SYNC_PULSE = 6;          // vertical sync
   parameter  V_BP = 29;                 // vertical back porch

   // horizontal: 1344 pixels total
   // display 1024 pixels per line
   logic hblank,vblank;
```

```
   logic hsyncon,hsyncoff,hreset,hblankon;
   assign hblankon = (hcount_out == (DISPLAY_WIDTH -1));
   assign hsyncon = (hcount_out == (DISPLAY_WIDTH + H_FP - 1));
//1047
   assign hsyncoff = (hcount_out == (DISPLAY_WIDTH + H_FP +
H_SYNC_PULSE - 1));  // 1183
   assign hreset = (hcount_out == (DISPLAY_WIDTH + H_FP +
H_SYNC_PULSE + H_BP - 1));  //1343

   // vertical: 806 lines total
   // display 768 lines
   logic vsyncon,vsyncoff,vreset,vblankon;
   assign vblankon = hreset & (vcount_out == (DISPLAY_HEIGHT -
1));   // 767
   assign vsyncon = hreset & (vcount_out == (DISPLAY_HEIGHT +
V_FP - 1));  // 771
   assign vsyncoff = hreset & (vcount_out == (DISPLAY_HEIGHT +
V_FP + V_SYNC_PULSE - 1));  // 777
   assign vreset = hreset & (vcount_out == (DISPLAY_HEIGHT +
V_FP + V_SYNC_PULSE + V_BP - 1)); // 805

   // sync and blanking
   logic next_hblank,next_vblank;
   assign next_hblank = hreset ? 0 : hblankon ? 1 : hblank;
   assign next_vblank = vreset ? 0 : vblankon ? 1 : vblank;
   always_ff @(posedge vclock_in) begin
      if(reset) begin
```

```
      hcount_out <= 0;
      vcount_out <= 0;
      hblank <= 0;
      hsync_out <= 1;
      vblank <= 0;
      vsync_out <= 1;
      blank_out <= 0;
    end else begin
      hcount_out <= hreset ? 0 : hcount_out + 1;
      hblank <= next_hblank;
      hsync_out <= hsyncon ? 0 : hsyncoff ? 1 : hsync_out;  //
active low

      vcount_out <= hreset ? (vreset ? 0 : vcount_out + 1) :
vcount_out;
      vblank <= next_vblank;
      vsync_out <= vsyncon ? 0 : vsyncoff ? 1 : vsync_out;  //
active low

      blank_out <= next_vblank | (next_hblank & ~hreset);
    end
  end
endmodule

`default_nettype wire
`default_nettype none
```

```
module debounce (input wire reset_in, clock_in, noisy_in,
                 output logic clean_out);
   parameter DEBOUNCE_COUNT = 1000000;
   logic [19:0] count;
   logic new_input;

   always_ff @(posedge clock_in)
     if (reset_in) begin
        new_input <= noisy_in;
        clean_out <= noisy_in;
        count <= 0; end
     else if (noisy_in != new_input) begin new_input<=noisy_in;
count <= 0; end
     else if (count == DEBOUNCE_COUNT) clean_out <= new_input;
     else count <= count+1;


endmodule

`default_nettype wire
```