6.111 Final Project

KWala Electronics Project

walaa - kennethc

Overview (walaa)

Our project intended to replicate the Tiger Electronics handheld LCD screen games that were popular in the 1990's. The device features a set number of buttons to control the game and an LCD screen with fixed character pixels. Each game would be based on a different movie or cartoon, and would be specifically for that set of characters and levels. The games also included music and a fixed, printed background image that lays behind the LCD screen.

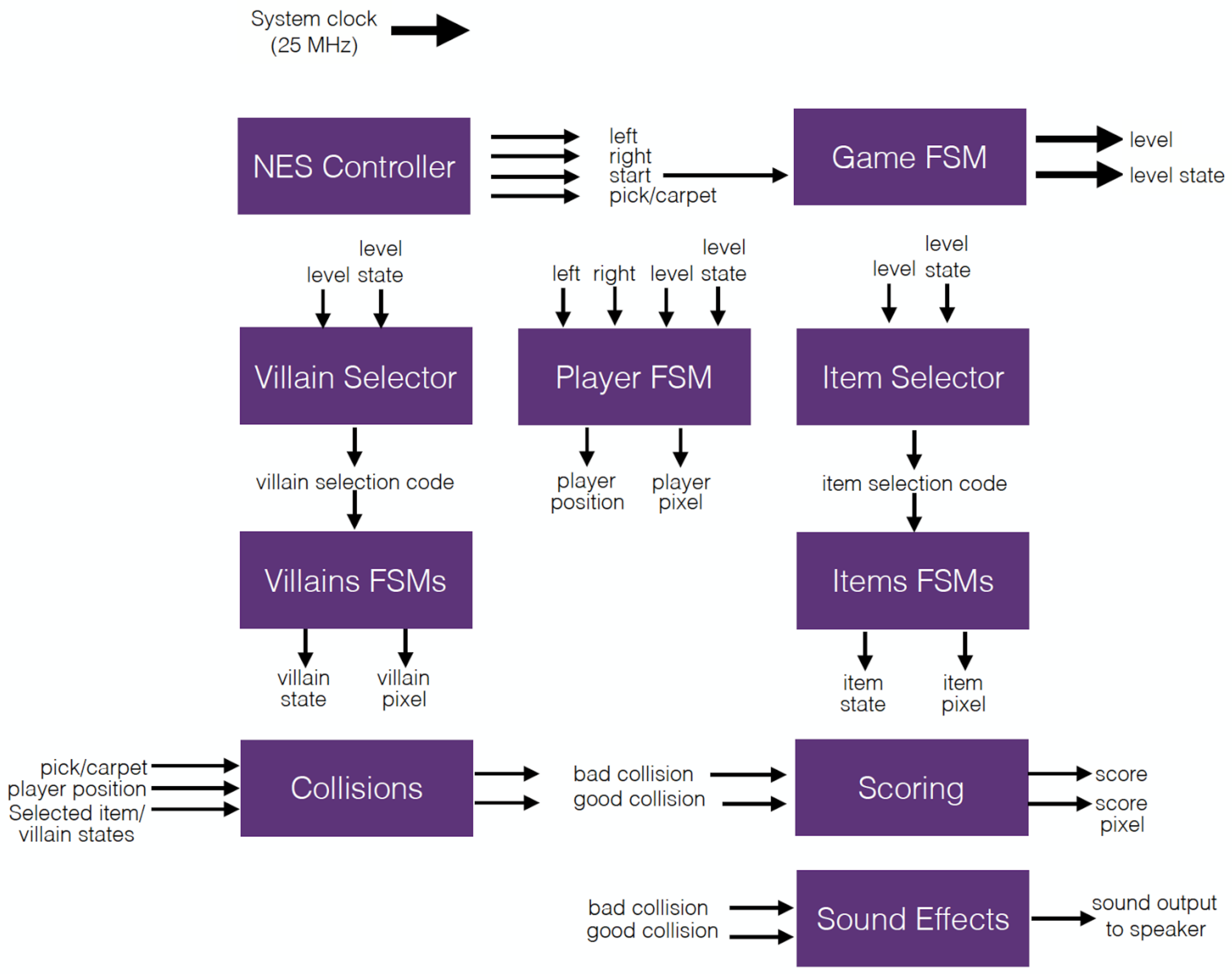
Our aim was to replicate not only one specific copy of the game, but produce a generic FSM that can be used with different specific verilog modules and graphics to be able to produce a number of these games, as they seem to follow the same FSM formula, have the same number of inputs, the same scoring, and similar level patterns.

In the end, we were able to produce one fully functional aladdin replica of the Tiger Electronics version, complete with graphics, music, and scoring, enhanced with: Nintendo NES controller input, sound effects, and a moving background.

Listing of Verilog Modules:

- Player FSM
- NES Controller FSM
- Modularity Support FSMs:
  - Game FSM
  - Villain Selector
  - Item Selector
- Villain FSMs:
  - Imperial Guard
  - Rocks
  - Snake/Jafar
- Item/Character FSMs:
  - Monkey/Abu
  - Apple
  - Carpet
  - Sword
- Scoring:
  - Score Unit
  - Score Graphics
- Collision Detection
  - Good Collision
  - Bad Collision
- Sound:
  - Sound Effects
  - Music
- Clock and timer Modules
  - 60 Hz Clock
  - 1 Hz Clock
  - 6 us timer
  - 12 us timer
  - Custom value seconds timer
- Character Graphics
  - Aladdin
  - Monkey/Abu
  - Imperial Guard/Jafar
  - Magic Carpet
- Other Graphics
  - Level dividers/ cover screens
  - Moving background

Block Diagram (walaa)



Game Logic (walaa)

In order to support multiple games using the same game logic, I decided to start out by making everything modular. As such, I decided to separate the process using modules that would allow for simpler integration of new games into the logic. These modules include the Selectors: Villain Selector and Item Selector, and the collision detection modules: Good Collision and Bad Collision. I also opted to separate different tasks into different modules rather than writing super-modules that did a sequence of different steps. I also parametrized many values in different modules in order to simplify modification if needed in the future, and in case any of the parameters would be better suited as inputs in the case of multiple game implementations sharing the same game logic.

Collision Logic (walaa)

The collision modules output boolean states of whether there is a good or bad collision detected. It is very concise and simple as a result of the modularity of the character and item FSMs. I designed each of those FSMS to have a 3-bit state. The first states can indicate something like an item's movement across the screen or a villains attack. 3'b111, however is reserved for the item when it is in the state in which it can be collided with. That way, all a Bad Collision module needs to do is check whether the player is in the standing position (attackable) and the villain is in the attacking position (3'b111). As for the Good Collision module, it requires player input as the player has to pick the item or interact with it using the pick/carpet button, in which case the additional check of whether the pick/carpet button on the player's controller is pressed.

Selector Logic (walaa)

Similarly to the collision units, as a result of the modularity of the logic, selector modules are fairly simple. Depending on the level of the game and the state of the level, both outputted from the game FSM, the selector can output the item or villain code. A check of whether the selected code matches each certain item or villain's particular code tells each FSM whether it is on or off.

Scoring Logic (walaa)

Score depends on the output of the collision unit: score goes higher on a good collision and lower on a bad one. The minimum is 0 and the score doesn't overflow.

Bug: Score would be changed (incremented or decremented) continuously, every clock cycle, as long as a collision boolean is high. I had to use a flag to denote that the collision has stopped, and only count the score once at each detected collision assertion.

FSMs (walaa)

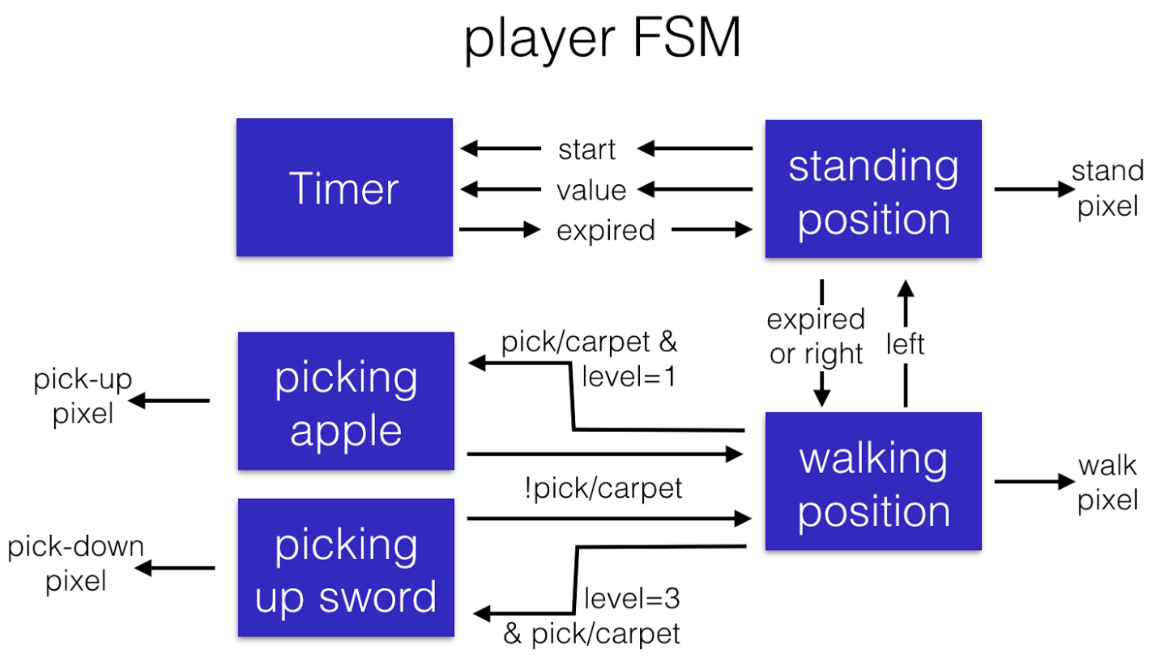
The game is comprised of many(, many) quite small and fairly simple FSMs so as not to create one giant complex FSM to contain all the different possibilities of the state of the game as a whole which would be impossible to debug and very difficult to modify and reason about.

There are three items and three villains, each its own FSM and verilog module. Almost every FSM required its own timer in order to simulate the (choppy) movement of that particular item or character on the LCD screen space.

Here are a few of the FSMs explained in more detail below:

Player FSM (walaa)

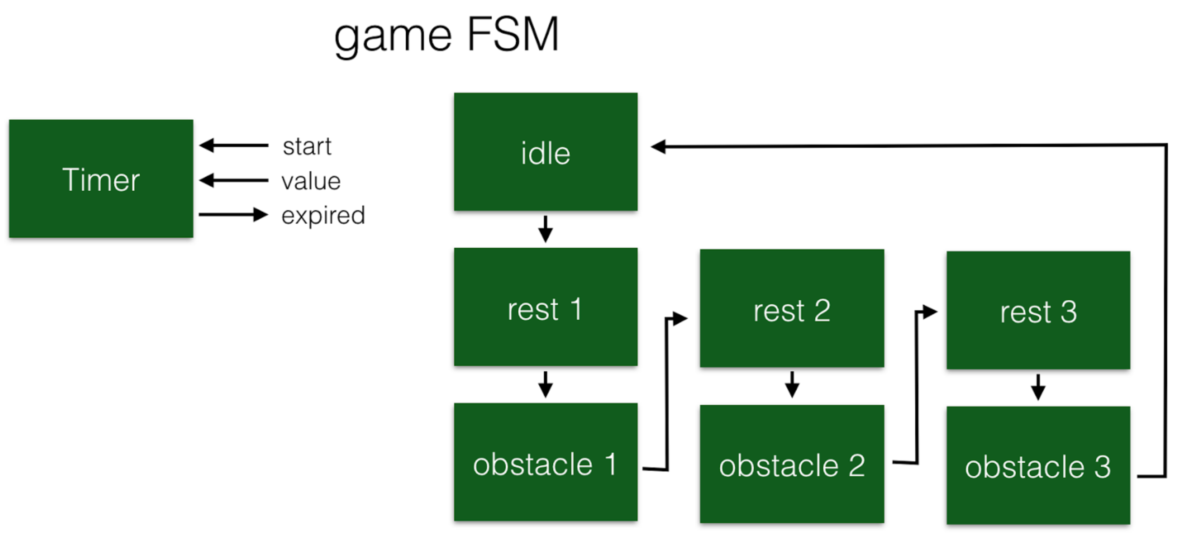
The player FSM, controls the state of the player in the game. There are two main states: standing and walking. While in the walking state, the player can transfer into the pick state which produces different images depending on the level. When the player moves into the stand state (which is the safe state from all villain attacks), a timer of 4 seconds is started. If the player states in this safe state for too long, he is forced forward into the running position. Here is a clarifying block diagram:



In truth, the walking position represents two states nested within the walking state. Since the player character has to appear as though it is moving, we have to switch between two images of him walking. Therefore we need to time it and change back and forth to show this. The above depiction also simplifies the FSM as the pixel assignment is not done quite so simply. Instead of directly outputting the pixel depending on the state, as the state can change too suddenly and uncontrollably within the system clock domain, I chose to assign an enable variable for each depiction of the character. Using this strategy, the pixel assignment statement included all the possible pixels ANDed with their enable signals, and OR'ed together.

Game FSM (walaa)

The game FSM is the simplest of the FSMs because it is the highest level one that doesn't deal with almost any of the complexities of the actual game logic itself.



Also heavily simplified in the block diagram above, the FSM is still not quite that straightforward. The complexity in it is mainly to replicate the Tiger Electronics game that we had physical access to as well as enhance the game-like experience. Each state transition between states (except the one from "idle" to "rest 1") is caused by the expiration of the timer. When entering a new state, the timer is told to start with the relevant (parameterized) value for that particular stage of the level. When transitioning between "obstacle 3" and "idle" states, the level has to be incremented. When in the idle state, the FSM awaits input "start" from the player to move on to the next state and actually start the game. When in state "idle," everything else in the game logic and every other FSM including the player, all items, characters, scoring, and collision detection is paused.

Level Cover Graphics (walaa)

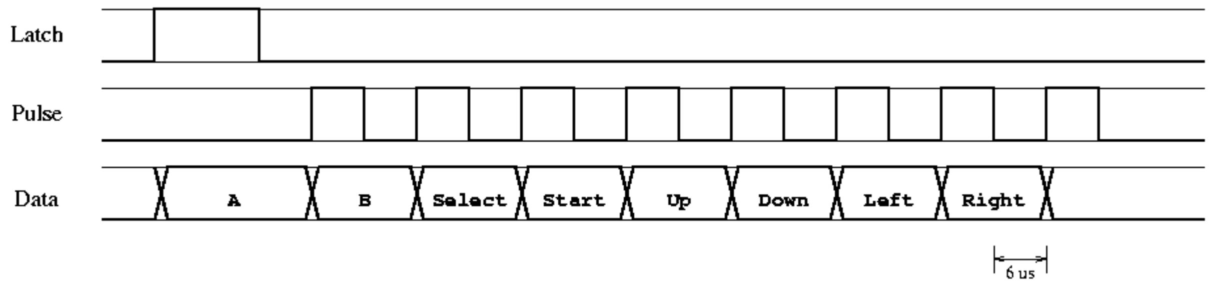
This is the only module that is active when the state of the level is "idle." The displayed image on the screen simply shows the current level of the game that the player is on.

NES Controller (walaa)

After developing the game logic and FSMs using control inputs from the FPGA buttons, I started to integrate the Nintendo NES Controller for input into the game. A super helpful resource with the pinouts of the controller is this webpage: https://tresi.github.io/nes/

In my module, I created a 60Hz clock on the positive edge of which I would sent a Latch signal to the controller, asking it to record the state of button presses. This signal stays high for 12us, so I created a timer that gets called at each latch assertion. I also created a 6us timer that allows to send the eight pulses to the controller with a period of 12us and a duty cycle of 50%. I read the data at the negative edge of the latch and first seven pulse signals, because it is transitioning at the rising edges of the pulse signals. I use a shift register to store the data in as I read another input, and once all data is recorded, I output the inverted state of the buttons on the controller, as they are negative true.

I powered the controller with a +5V supply and ground, and left the wires that don't go anywhere disconnected. The signals sent and received are shown clearly in this graph:



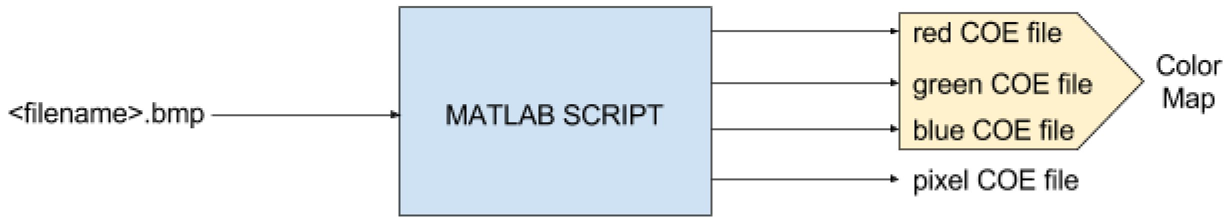
Bug: I initially misread/misunderstood the signals and thought that all we send is the latch, and wait for the controller to send us the pulses upon whose negative edges we read the data. I thoroughly and extensively probed my circuit and found the same odd behavior from the controller and ultimately believed the controller was broken. Once I reread the above linked webpage enough times to catch my misunderstanding and I started sending out the eight pulses, the module worked fine

Sound Effects (walaa)

This module took as main input the booleans good collision and bad collision. On good collision, it produces a high frequency sound, and on bad collisions it produces a low frequency sound. It therefore holds two constants, one for each frequency threshold. When a collision is asserted, the module outputs a 1'b1 at the threshold frequency, and increments or decrements it appropriately in order to sweep high or low frequencies depending on the collision that triggered the sound effect. This output is then sent to the speakers in the same way we did in lab4 car alarm.

Graphics (kennethc)

Each image that was used for the project was processed using a MATLAB script that created 4 different COE files.



These COE files were used to format Block RAM on the FPGA as single port ROM tables. Each bitmap file contained `x` number of bits per pixel, meaning one image would take `PICTURE_WIDTH*PICTURE_HEIGHT*x` bits of memory. For this project, each image was 4 bits, thus allowing for a maximum of 16 colors in each image. This means that in the pixel COE file, there were `PICTURE_WIDTH*PICTURE_HEIGHT` lines with 4 bits per line. The BRAM with this COE file does the following:

- INPUT: `image_addr` → the pixel of interest. If we need the 4 bits of information that encode the colors for pixel 0 (upper leftmost pixel), then `image_addr = 0` (the very first line in this BRAM module)
- OUTPUT: `image_bits` → the 4 bits of information that encode the colors for the pixel `image_addr`. This was outputted in 1 clock cycle.

Once the `image_bits` had been calculated, they were fed into the red, green, and blue COE files which collectively made a color map for the image(s). These COE files were unique for each image, and they did the following:

- INPUT: `image_bits` → see above description
- OUTPUT: `r, g, b` → these are registers that contain the RGB values (respectively) for a particular pixel. These RGB values will output to `VGA_R`, `VGA_G`, `VGA_B` to generate the right color for VGA output. The MATLAB script created an appropriate mapping from 4 bits of information to 8 bits of red, 8 bits of green, and 8 bits of blue.

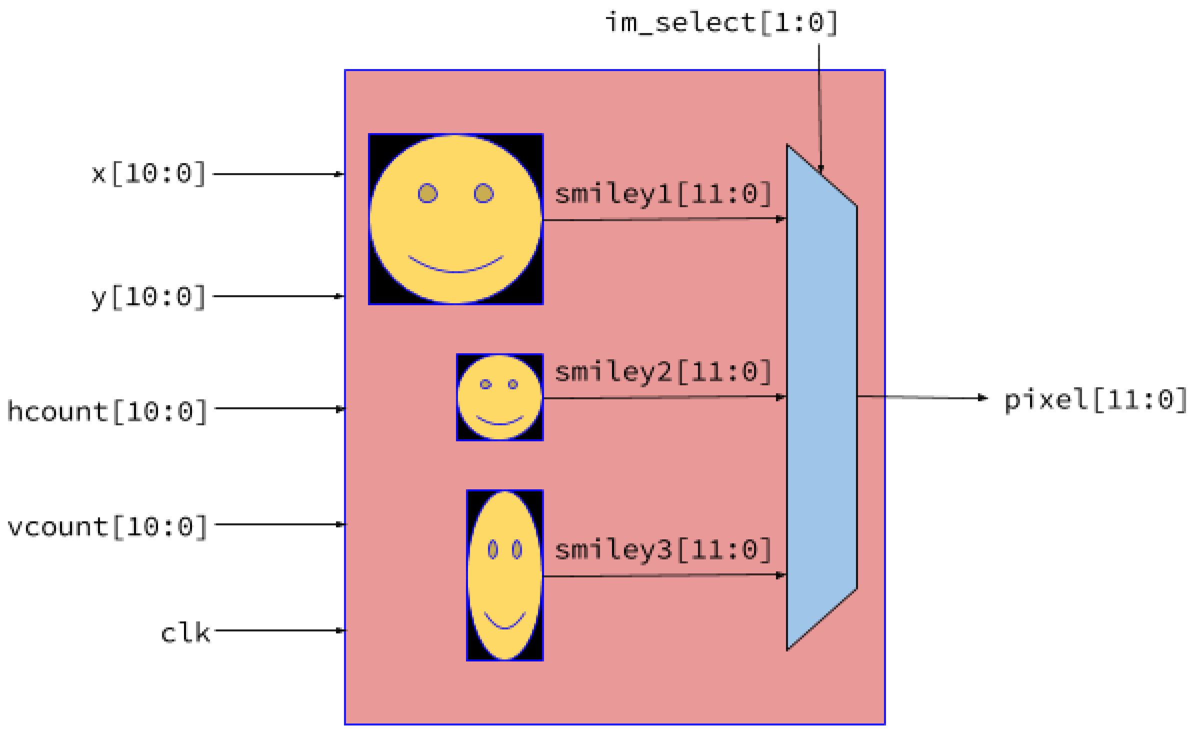
VGA requires 24 bits of color (`[23:16]`, `[15:8]`, `[7:0]`). However, the Nexys 4 can only handle 4 bits per color (12 bits total). Thus within each color map, the highest 4 bits for each color were kept.

Generating a color map for each image used in the project would have taken up a lot of BRAM. To cut down on memory usage, I

1) created a sprite sheet that included every image that would be used in the project,
2) limited the number of colors on the sprite sheet to 16 colors (4 bits),
3) created a single color map using the sprite sheet, and
4) cropped each individual image and created a pixel COE file for each one.

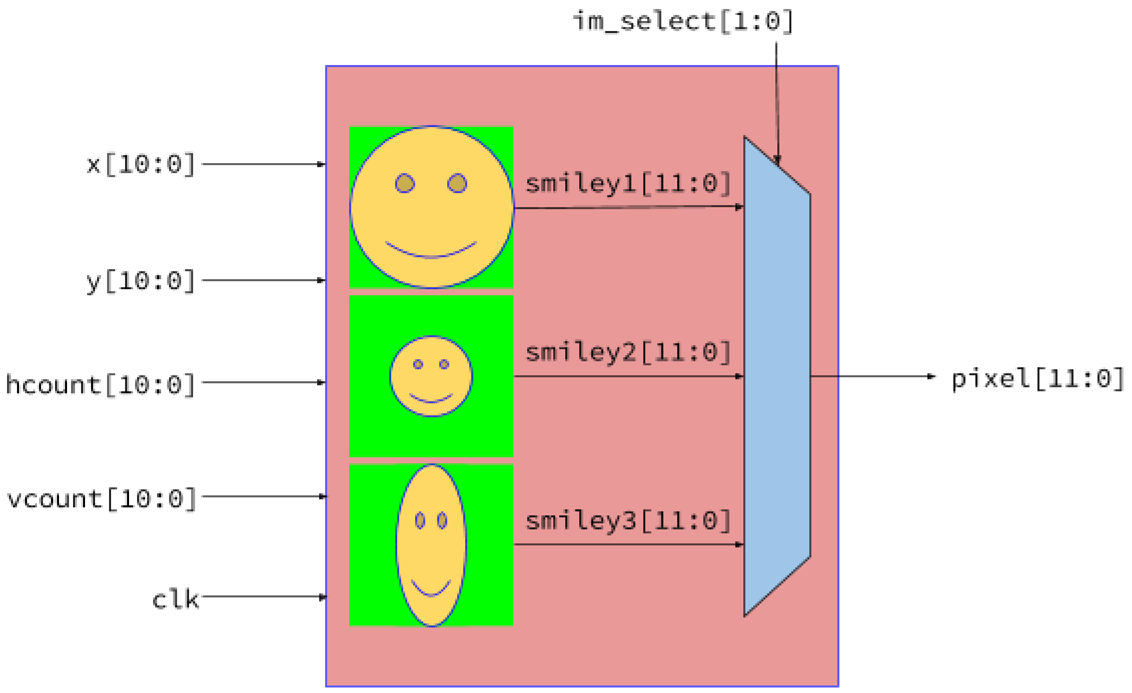
This method allowed for me to dedicate the vast majority of the BRAM to storing the images since I had a universal color map that applied to all the sprites we used. Since the color map was only 16 lines with 12 bits per line, it was formatted using `case(image_bits)` and combinational logic to give the right color within the same clock cycle it took to access the pixel BRAM. Overall, only 2 sets of color maps were used (the second color map was used to generate the koala image at the beginning and the end of the game). Refer to the Appendix to see the sprite sheet as well as all of the sprites used.

Each sprite had a separate module for drawing it. However, for certain characters (Aladdin, Abu, Carpet, Villains), the sprite could take on several different positions. In Aladdin's case, there were two different images to generate a running motion, one image for standing, one image for grabbing things below (swords), and one image for grabbing things above (apples). That's 5 different images just for one sprite! A simple way to organize all the sprites for one character was to have one module encapsulate all the sprites for a particular character and multiplex them to determine which image is drawn onto the screen.



This is a module dedicated to drawing different forms of the smiley sprite where `smiley1`, `smiley2`, and `smiley3` contain the 12 bit color info for pixels of the 1st, 2nd, and 3rd pictures respectively. `im_select` determines which picture will be outputted from this module.

With inputs `x`, `y`, and `im_select` we could choose which image of our character to draw and where we wanted it to be drawn. These "image muxes" used combinational logic to yield an output. The `clk` input is used to run the sequential logic that draws the images. The one problem with this was that the images for one character could have different sizes. To fix this problem, all the images that applied to one character were given the same width and height, and the original image itself was shifted as necessary. This way, there was less of a need to change `x` and `y` to account for differences in size and positioning.



This has it to where the original images were all shifted to the middle of the larger window being used to represent this character. Now we can change the image displayed without having to change `x` or `y` (unless we want our sprite to move elsewhere)

Lastly, we didn't want the background color of the image to display and interfere with the background image of the game. For each image I used a green background color (similar to the one above). Within each image drawing module, I added logic to create a dark pixel (`12'b0`) wherever this particular green colored pixel was encountered. Here is an example output if we were to select `smiley1`:

Original Image

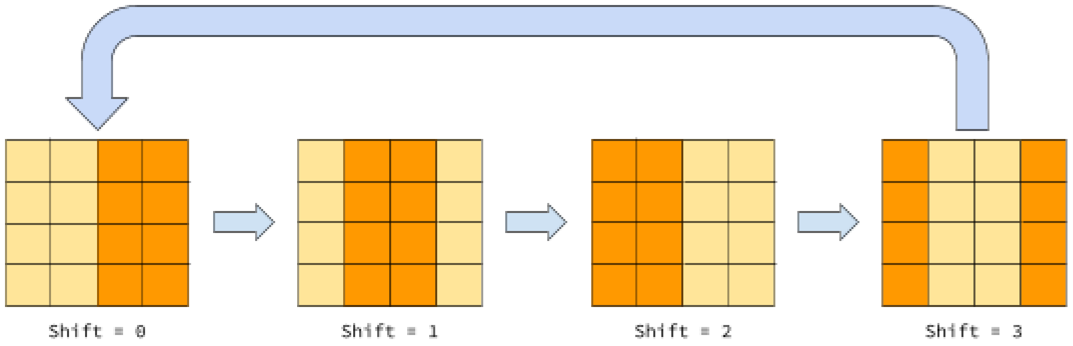
Displayed Image

Then, a few more lines of code were added to draw the pixels of the background image over any dark pixel that is currently on the screen. At a given pixel, if no color from another pixel "inhabits" that area, then the background would fill in those gaps. This method works a lot like a green screen works. This particular green was used because it was a color that was extremely different compared to all the other colors present on the sprite sheet. Thus, targeting these pixels and making them dark pixels did not alter any other part of any other image.

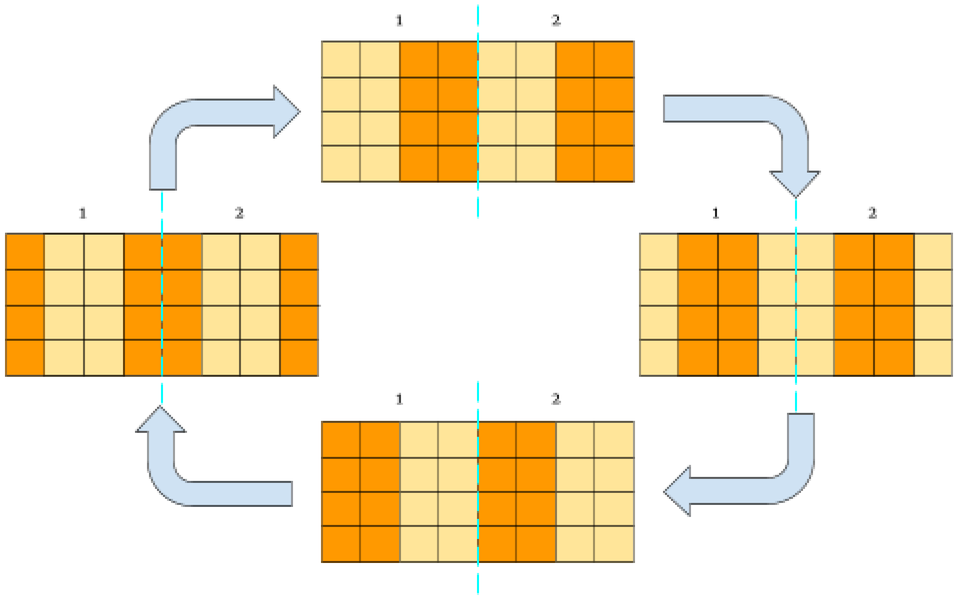
The background module was a little different than the other image generating modules. In order to give the illusion of moving forward, I decided to make the background scroll. For this it is necessary to look at the equation used to calculate the next `image_addr`:

```
image_addr <= (hcount - x) + (vcount - y)*WIDTH
```

This is the calculation to find the next pixel to look at. If `x` is replaced by `x - shift`, then every pixel location displayed is shifted to the left by `shift` amount. Here's an example of a shift by 1:

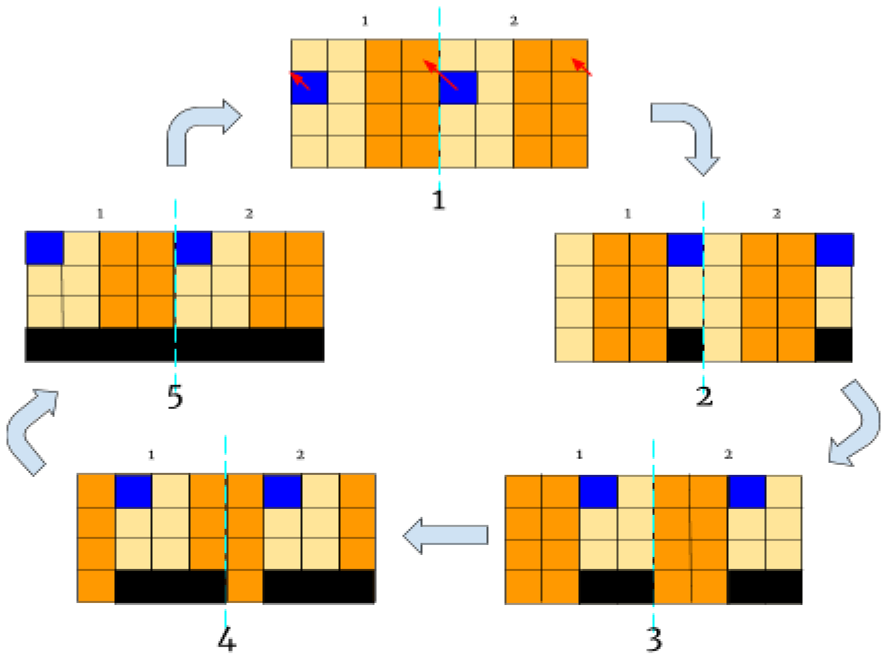


As demonstrated above, the pixels that were on the leftmost edge of the image wrapped back around to the rightmost side. Thus in this case, after 4 shifts, the original image would return.



For the implementation of the game, 2 instances of the background were created side by side and after a second they shifted by the same amount. This created the illusion of one large image scrolling across the screen, analogous to the diagram above.

Unfortunately, there was a small bug that I could not find a solution to. Instead of the background doing as illustrated, it did the following (assume the blue pixel to be a specific detail in the background image):



All pixels that had wrapped to the back of the image had been shifted up by one and continued to shift to the left in this same matter. Once frame 5 had been reached, the image then sat back down to go back to frame 1, and it repeated. Where random dark pixels occured may have been pixels where there was an error in calculating `image_addr` with the added shift, causing there to be no color at all. This suspicion was not and has not been confirmed.
To finish the background, a solid color matching with the background was added to the very bottom, providing a space for all the sprites to “stand” on.

Music **(kennethc)**

The music used in a game is a short excerpt from the Super Mario Theme from the iconic game *Super Mario Bros*. In order to generate the music, I had to find a table that listed which tones/frequencies were played, what order they were played in, and the duration of each frequency. The information that was used was found at https://wiki.mikrotik.com/wiki/Super_Mario_Theme. Here is a snippet of the format the information was in:

```
:beep frequency=510 length=100ms;
:delay 450ms;
:beep frequency=380 length=100ms;
```

```
        :delay 400ms;
        :beep frequency=320 length=100ms;
        :delay 500ms;
```

There are 33 lines of information like this that encoded the music. The next steps were to 1) calculate the number of cycles necessary to calculate each frequency, 2) calculate the number of cycles necessary to create each duration, and 3) convert that information into a binary string. For step 1, I calculated the number of cycles for a given frequency using this equation:

$$\text{\# of cycles} = f_{\text{clock}} * T_{\text{tone}} = 25{,}000{,}000 \text{ Hz} * 1/f_{\text{tone}}$$

Using a similar formula, I could find the number of cycles necessary to approximate duration of each tone:

$$\text{\# of cycles} = f_{\text{clock}} * \text{duration (in seconds)}, \quad 1 \text{ s} = 10^{-3} \text{ ms}$$

If we look at the first line in the example above, the number of cycles necessary to generate a 510 Hz tone would be `49019` (as a truncated integer), or `1011111101111011` in binary. The duration for this tone would be `2500000` cycles, or `1001100010010110100000` in binary. All delays we given a tone of 0 Hz which was encoded by `0`s in binary. This information was then used to create a ROM table.

I decided that each entry of the ROM table would contain information for both the frequency and the duration. After calculating the largest number of cycles for lowest frequency and the largest number of cycles for the longest duration, it was determined that at least 24 bits were required to encode duration and at least 17 bits were required to encode frequency. This meant all 33 lines of the ROM table had 41 bits of information. Extra `0`s were added when necessary in order to make sure each line had 42 bits. For instance, the resulting line in the ROM table for the calculation above (`:beep frequency=510 length=100ms;`) would be `24’b01011111101111011001001100010010110100000`.

The output of this module was logically `OR`’d with the output of the SFX module which was then fed into an amplifying circuit that fed into a speaker.

Verilog Appendix

(walaa modules)

```
module Game_FSM(
    input clk,
    input rst,
    input start, //button
    input timer_expired,
    output reg start_timer,
    output reg [6:0] timer_value,
    output reg ongoing, //boolean
    output reg [2:0] level_state,
    output reg [2:0] game_level
    );

//define level states
wire [2:0] idle, start_level, obstacle_1, rest_1, obstacle_2, rest_2,
obstacle_3, rest_3;
assign idle = 3'b0;
assign start_level = 3'b1;
assign obstacle_1 = 3'd2;
assign rest_1 = 3'd3;
assign obstacle_2 = 3'd4;
assign rest_2 = 3'd5;
assign obstacle_3 = 3'd6;
assign rest_3 = 3'd7;

parameter obstacle_time = 7'd4;
parameter rest_time = 7'd5;
parameter start_level_time = 3;
reg start_flag;

always @(posedge clk) begin
    start_flag <= !start;
    //initialize game FSM
    if (rst) begin
      ongoing <= 0;
      game_level <= 0;
      level_state <= idle;
    end
    //level zero before the game starts
    else if (game_level == 0 & start) begin
        game_level <= 1;
        ongoing <= 1;
    end
    //level four after the game ends
    else if (game_level == 3'd4) begin
        ongoing <= 0;
    end

    //update game states appropriately
    else if (ongoing) begin
      case (level_state)

      idle: begin
              //actually start the level when start is pressed
              if (start & start_flag) begin
                level_state <= start_level;
                start_timer <= 1;
                timer_value <= start_level_time; //3 seconds of nothing
happening in the level
              end
            end

      start_level: begin
                    //when start_level timer expires, go to the next game
state
                    if (timer_expired) begin
                      level_state <= obstacle_1;
                      start_timer <= 1;
                      timer_value <= obstacle_time;//7'b1111111; //run the
next obstacle for ten seconds
                    end  else start_timer <= 0;
                  end

      obstacle_1: begin
                    //when obstacle timer expires, go to the next level in the
game
                    if (timer_expired) begin
                      level_state <= rest_1;
                      start_timer <= 1;
                      timer_value <= rest_time;
                    end  else start_timer <= 0;
                  end

      rest_1:begin
                if (timer_expired) begin
                level_state <= obstacle_2;
                start_timer <= 1;
                timer_value <= obstacle_time;
              end  else start_timer <= 0;
            end
      obstacle_2: begin
                    //when obstacle timer expires, go to the next level in the
game
                    if (timer_expired) begin
                      level_state <= rest_2;
                      start_timer <= 1;
                      timer_value <= rest_time;
                    end  else start_timer <= 0;
                  end

      rest_2:begin
                if (timer_expired) begin
                level_state <= obstacle_3;
                start_timer <= 1;
                timer_value <= obstacle_time;
              end  else start_timer <= 0;
            end

      obstacle_3: begin
                    //when obstacle timer expires, go to the next level in the
game
                    if (timer_expired) begin
                      level_state <= rest_3;
                      start_timer <= 1;
                      timer_value <= rest_time;
                    end  else start_timer <= 0;
                  end

      rest_3:begin
                if (timer_expired) begin
                level_state <= idle;
                game_level <= game_level+1;
              end  else start_timer <= 0;
            end

      endcase
    end else begin
        game_level <= 0;
        level_state <= idle;
    end
end

endmodule
```

```
module Player_FSM(
    input clk,
    input rst,
    input ongoing,
    input [2:0] level_state,
    input [1:0] game_level,
    input pick_carpet,
    input left, //button
    input right, //button
    input timer_expired,
    input [9:0] hcount,
    input [9:0] vcount,
    input vsync,
    output reg apple_throw,
    output reg [11:0] pixel,
    output reg start_timer,
    output reg [6:0] timer_value,
    output reg player_position, //player has only two potential positions
    output reg timer_flag
    );

//define level states
    wire [2:0] idle, start_level, obstacle;
    assign idle = 3'b0;
    assign start_level = 3'b1;
    assign obstacle = 3'd2;

reg walk_1_pixel_enable, walk_2_pixel_enable, stand_pixel_enable,
pick_pixel_enable;

reg walking_flag;
//reg timer_flag;
reg pick_carpet_timer;
reg pick_carpet_flag;

//define blobs for each position of the player
wire [11:0] stand_pixel, walk_pixel_1, walk_pixel_2, pick_up_pixel,
pick_down_pixel, pick_pixel;

aladdin_mux standblob(.x(150),.y(350),.hcount(hcount),.vcount(vcount),
                     .im_sel(3'd0), .pixel(stand_pixel), .clk(clk));

aladdin_mux walkblob1(.x(250),.y(350),.hcount(hcount),.vcount(vcount),
                     .im_sel(3'd1), .pixel(walk_pixel_1), .clk(clk));

aladdin_mux walkblob2(.x(250),.y(350),.hcount(hcount),.vcount(vcount),
                     .im_sel(3'd2), .pixel(walk_pixel_2), .clk(clk));

aladdin_mux pickupblob(.x(250),.y(350),.hcount(hcount),.vcount(vcount),
                     .im_sel(3'd4), .pixel(pick_up_pixel), .clk(clk));

aladdin_mux pickdownblob(.x(250),.y(350),.hcount(hcount),.vcount(vcount),
                     .im_sel(3'd3), .pixel(pick_down_pixel), .clk(clk));

assign pick_pixel = (game_level==2'd1)? pick_up_pixel:((game_level==2'd3)?
pick_down_pixel: walk_pixel_1);

//always @(negedge vsync) begin
always @(posedge clk) begin
    //initialize player position to 1
    if (rst) begin
        player_position <= 1;
        timer_flag <= 1;
        walking_flag <= 1;
        pick_carpet_timer <= 1;
        pick_carpet_flag <= 0;
//        apple_throw <= 0;
    end

    if (ongoing) begin
      //udpate player position
      if (left)
            player_position <= 0; //left position

      if (right)
            player_position <= 1; //right_position

      //if player is in the safe position for too long, force him forward
      if (player_position == 0 && timer_flag) begin
         start_timer <= 1;
         timer_value <= 4'd4;
         timer_flag <= 0;
         walking_flag <= 1;
         pick_carpet_timer <= 1;
      end

      else if (player_position == 0 && timer_expired) player_position <= 1;

      else if (player_position == 1) begin
              timer_flag <= 1;
              start_timer <= 0;
              pick_carpet_flag <= 1;
            end
```

```
  else start_timer <= 0;

  casex({player_position, pick_carpet & game_level != 2'd2})
  2'b0X: begin
       stand_pixel_enable <= 1;
       walk_1_pixel_enable <= 0;
       walk_2_pixel_enable <= 0;
       pick_pixel_enable <= 0;
     end

  2'b10: begin
        if (walking_flag) begin
            timer_value <= 4'd1;
            start_timer <= 1;
            walking_flag <= 0;
            if (stand_pixel_enable | pick_pixel_enable) begin
                pick_pixel_enable <= 0;
                walk_1_pixel_enable <= 1;
                walk_2_pixel_enable <= 0;
                stand_pixel_enable <= 0;
            end else if (walk_1_pixel_enable) begin
                walk_1_pixel_enable <= 0;
                walk_2_pixel_enable <= 1;
            end else if (walk_2_pixel_enable) begin
                walk_1_pixel_enable <= 1;
                walk_2_pixel_enable <= 0;
            end
        end else start_timer <= 0;
        if (timer_expired) begin
            start_timer <= 0;
            walking_flag <= 1;
        end
     end

  2'b11: begin
    pick_pixel_enable <= 1;
    stand_pixel_enable <= 0;
    walk_1_pixel_enable <= 0;
    walk_2_pixel_enable <= 0;
  end
  endcase
end else begin
    player_position <= 1;
    walk_1_pixel_enable <= 1;
    timer_flag <= 1;
    walking_flag <= 1;
    pick_carpet_timer <= 1;
    pick_carpet_flag <= 0;
    end

    if (level_state != idle)
        pixel <= (stand_pixel & {12{stand_pixel_enable}}) |
                    (walk_pixel_1 & {12{walk_1_pixel_enable}}) |
                    (walk_pixel_2 & {12{walk_2_pixel_enable}}) |
                    (pick_pixel & {12{pick_pixel_enable}});
end
endmodule

module Bad_Collision(
    input clk,
    input rst,
    input player_position,
    input [2:0] villain_state,
    output reg collided //boolean
    );

always @(posedge clk) begin
    //if player is in the collision position, check state of item
    //item must also be in a "collision state"
    if (player_position & (villain_state == 3'b111)) collided <= 1;
    else collided <= 0;
end
endmodule


module Good_Collision(
    input clk,
    input rst,
    input pick_or_carpet,
    input player_position,
    input [2:0] item_state,
    output reg collided //boolean
    );

always @(posedge clk) begin
    //if player is in the collision position, check state of item
    //item must also be in a "collision state"
    if (player_position & (item_state == 3'b111) & pick_or_carpet)
        collided <= 1;
    else
        collided <= 0;
end

endmodule
```

```
`timescale 1ns / 1ps

module item_selector(
    input clk,
    input rst,
    input ongoing,
    input [2:0] level_state,
    input [1:0] game_level,
    output reg [2:0] item
    );


    //define the level states of each level
    wire [2:0] idle, start_level, obstacle_1, rest_1, obstacle_2, rest_2,
obstacle_3, rest_3;
    assign idle = 3'b0;
    assign start_level = 3'b1;
    assign obstacle_1 = 3'd2;
    assign rest_1 = 3'd3;
    assign obstacle_2 = 3'd4;
    assign rest_2 = 3'd5;
    assign obstacle_3 = 3'd6;
    assign rest_3 = 3'd7;

    //define each item
    wire [2:0] apples, carpet, sword;
    assign apples = 3'd1;
    assign carpet = 3'd2;
    assign sword = 3'd3;

always @(posedge clk) begin
    if (rst) item <= 3'd0;

    if (ongoing & (level_state == rest_1 | level_state == rest_2 | level_state
== rest_3)) begin
      case(game_level)
      3'd1: item <= apples;
      3'd2: item <= carpet;
      3'd3: item <= sword;
     endcase
    end else item <= 3'd0;
end

endmodule


module apples(
    input clk,
    input item_on,
    input picked,
    input timer_expired,
    input player_position,
    input [9:0] hcount,
    input [10:0] vcount,
    output reg start_timer,
    output reg [6:0] timer_value,
    output reg [2:0] state,
    output reg [11:0] pixel
    );

    reg timer_flag;
    reg out_of_screen;

    reg [10:0] x;
    wire [11:0] apple_pixel;

    gen_apple_bowl apple(.x(x),.y(320),.hcount(hcount),.vcount(vcount),
                         .pixel(apple_pixel), .pixel_clk(clk));

    always @(posedge clk) begin
        if (item_on && !out_of_screen) begin
            pixel <= apple_pixel;

            if (picked) begin
                out_of_screen <= 1;
            end

            else if (player_position) begin
                if (timer_flag) begin
                    start_timer <= 1;
                    timer_value <= 1;
                    timer_flag <= 0;
                end else start_timer <= 0;
                if (timer_expired) begin
                    state <= (state==2)? 3'b111:state + 1;
                    out_of_screen <= state == 3'b111;
                    timer_flag <= 1;
                end
            //if the player is not in the running position, allow the movement
timer to restart
            end else
                timer_flag <= 1;
        //if the pixel for the item isnt on
        end else if (!item_on) begin
            pixel <= 0;
            state <= 0;
            timer_flag <= 1;
            out_of_screen <= 0;
        end else pixel <= 0;

        case (state)
        3'd0: x <= 850;
        3'd1: x <= 600;
        3'd2: x <= 450;
        3'b111: x <= 325;
        endcase
    end

endmodule


module sword(
    input clk,
    input item_on,
    input picked,
    input timer_expired,
    input player_position,
    input [9:0] hcount,
    input [10:0] vcount,
    output reg start_timer,
    output reg [6:0] timer_value,
    output reg [2:0] state,
    output reg [11:0] pixel
    );

    reg timer_flag;
    reg out_of_screen;

    reg [10:0] x;
    wire [11:0] sword_pixel;
    //    blob #(.WIDTH(100),.HEIGHT(20),.COLOR(12'h666)) // white
    //        sword(.x(x),.y(400),.hcount(hcount),.vcount(vcount),
    //                 .pixel(sword_pixel));

        gen_sword sword(.x(x),.y(400),.hcount(hcount),.vcount(vcount),
                        .pixel_clk(clk), .pixel(sword_pixel));

    always @(posedge clk) begin
        if (item_on && !out_of_screen) begin
            pixel <= sword_pixel;

            if (picked) begin
                out_of_screen <= 1;
            end
            if (player_position) begin
                if (timer_flag) begin
                    start_timer <= 1;
                    timer_value <= 1;
                    timer_flag <= 0;
                end else start_timer <= 0;
                if (timer_expired) begin
                    state <= (state==2)? 3'b111:state + 1;
                    out_of_screen <= state == 3'b111;
                    timer_flag <= 1;
                end
            //if the player is not in the running position, allow the movement
timer to restart
            end else
                timer_flag <= 1;
        //if the pixel for the item isnt on
        end else if (!item_on) begin
            pixel <= 0;
            state <= 0;
            timer_flag <= 1;
            out_of_screen <= 0;
        end else pixel <= 0;

        case (state)
        3'd0: x <= 850;
        3'd1: x <= 600;
        3'd2: x <= 400;
        3'b111: x <= 200;
        endcase
    end

endmodule



module Flying_Carpet(
    input clk,
    input pick_carpet,
    input timer_expired,
    input [1:0] game_level,
    input [2:0] level_state,
    input [9:0] hcount,
    input [10:0] vcount,
    output reg start_timer,
    output reg [6:0] timer_value,
    output reg [11:0] pixel
    );
    reg [1:0] state;

    wire [11:0] stand_carpet_pixel, fly_carpet_pixel, fall_carpet_pixel;
    wire [11:0] stand_carpet_pixel, fly_carpet_pixel, fall_carpet_pixel;


    carpet_mux stand_carpet(.x(200),.y(370),.hcount(hcount),.vcount(vcount),
                            .im_sel(2'd1), .pixel(stand_carpet_pixel),
.clk(clk));

    carpet_mux fly_carpet(.x(295),.y(230),.hcount(hcount),.vcount(vcount),
                          .im_sel(2'd0), .pixel(fly_carpet_pixel), .clk(clk));

    carpet_mux fall_carpet(.x(295),.y(410),.hcount(hcount),.vcount(vcount),
                            .im_sel(2'd0), .pixel(fall_carpet_pixel),
.clk(clk));

    wire [2:0] stand, flying, fall;
    assign stand = 3'b1;
    assign flying = 3'd2;
    assign fall = 3'd3;

always @ (posedge clk) begin
    if (game_level == 2'd2 & level_state != 0) begin
      case (state)
      stand:  pixel <= stand_carpet_pixel;
      flying: pixel <= fly_carpet_pixel;
      fall:   pixel <= fall_carpet_pixel;
      endcase

      if (pick_carpet & state == stand) begin
        state <= flying;
        start_timer <= 1;
        timer_value <= 1;
      end else if (timer_expired & state == flying) begin
        state <= fall;
        start_timer <= 1;
        timer_value <= 2;
      end else if (timer_expired & state == fall)
        state <= stand;
      else start_timer <= 0;
   end else begin
        pixel <= 0;
        state <= stand;
   end
end
```

```
endmodule

`timescale 1ns / 1ps

module monkey(
    input clk,
    input item_on,
    input player_position,
    input pick_carpet,
    input timer_expired,
    input [1:0] game_level,
    input [2:0] level_state,
    input [9:0] hcount,
    input [10:0] vcount,
    output reg start_timer,
    output reg [6:0] timer_value,
    output reg [2:0] state,
    output reg saved,
    output reg [11:0] pixel
    );

    wire [11:0] stand_monkey_pixel, walk_monkey_pixel_1, walk_monkey_pixel_2;
    reg [11:0] walk_monkey_pixel; reg walk = 0, walk_flag = 1;
    reg [10:0] y;
    //     blob #(.WIDTH(30),.HEIGHT(50),.COLOR(12'hFAA)) // red
    //         flymonkey(.x(300),.y(y),.hcount(hcount),.vcount(vcount),
    //                  .pixel(monkey_pixel));


    parameter monkey_height = 390;
    abu_mux flymonkey(.x(300),.y(y),.hcount(hcount),.vcount(vcount),
                      .im_sel(2'b0), .clk(clk), .pixel(stand_monkey_pixel));
    abu_mux
walk1monkey(.x(300),.y(monkey_height),.hcount(hcount),.vcount(vcount),
                        .im_sel(2'b1), .clk(clk),
.pixel(walk_monkey_pixel_1));
    abu_mux
walk2monkey(.x(300),.y(monkey_height),.hcount(hcount),.vcount(vcount),
                        .im_sel(2'd2), .clk(clk),
.pixel(walk_monkey_pixel_2));

    wire [2:0] stand, flying, fall;

    assign stand = 3'b1;
    assign flying = 3'b111; // if flying and pick_carpet, he has been saved
and

    reg done;
```

```
always @ (posedge clk) begin
    pixel <= (player_position & !item_on)? walk_monkey_pixel:
stand_monkey_pixel;

    if (level_state != 0) begin
      case (state)
      stand:  y <= monkey_height;
      flying: y <= 200;
      endcase
    end

    if (item_on & !done) begin
      walk_flag <= 1;
      if (state == stand) begin
        saved <= 0;
        state <= flying;
        start_timer <= 1;
        timer_value <= 2;
      end else if (pick_carpet & state==flying) begin
        saved <= 1;
        start_timer <= 1;
        timer_value <= 1;
      end else if (timer_expired & state == flying) begin
        done <= 1;
        state <= stand;
      end else start_timer <= 0;
   end else if (!item_on) begin
      state <= stand;
      done <= 0;
      if (player_position) begin
        case (walk)
        0: walk_monkey_pixel <= walk_monkey_pixel_1;
        1: walk_monkey_pixel <= walk_monkey_pixel_2;
        endcase
        if (timer_expired | walk_flag) begin
            walk <= !walk;
            start_timer <= 1;
            timer_value <= 1;
            walk_flag <= 0;
        end else start_timer <= 0;
      end else walk_flag <= 1;
   end else walk_flag <= 1;
end

endmodule


//rocks only fall if the player doesn't press pick_carpet at the right time
```

```
module Rocks(
    input clk,
    input pixel_on,
    input saved, // true if monkey has been saved
    input timer_expired,
    input [1:0] game_level,
    input [2:0] level_state,
    input [9:0] hcount,
    input [10:0] vcount,
    output reg start_timer,
    output reg [6:0] timer_value,
    output reg [2:0] state,
    output reg [11:0] pixel
    );

wire [11:0] hanging_rocks_pixel, falling_rocks_pixel, hitting_rocks_pixel,
rocks_pixel;
parameter x = 300;


module snake(
    input clk,
    input rst,
    input pixel_on,
    input timer_expired,
    input [9:0] hcount,
    input [10:0] vcount,
    output reg start_timer,
    output reg [6:0] timer_value,
    output reg [2:0] state,
    output reg [11:0] pixel
    );

//define the states of the guard
wire [2:0] snake, jaffar, start_attack, strike_attack;
assign snake = 3'd1;
assign jaffar = 3'd2;
assign start_attack = 3'd3;
assign mid_attack = 3'd4;
assign strike_attack = 3'b111;

reg timer_flag;

//define blobs for each state
wire [11:0] snake_pixel, jaffar_pixel, start_attack_pixel, mid_attack_pixel,
strike_attack_pixel, start_shoot, mid_shoot, strike_shoot;
```

```
badguy_mux jaffarblob(.x(450),.y(350),.hcount(hcount),.vcount(vcount),
                      .im_sel(3'd3), .pixel(jaffar_pixel), .clk(clk));

badguy_mux snakeblob(.x(350),.y(350),.hcount(hcount),.vcount(vcount),
                      .im_sel(3'd4), .pixel(snake_pixel), .clk(clk));

gen_fire startfire(.x(340),.y(340),.hcount(hcount),.vcount(vcount),
                   .pixel(start_shoot), .pixel_clk(clk));
gen_fire midfire(.x(325),.y(350),.hcount(hcount),.vcount(vcount),
                 .pixel(mid_shoot), .pixel_clk(clk));
gen_fire strikefire(.x(310),.y(360),.hcount(hcount),.vcount(vcount),
                 .pixel(strike_shoot), .pixel_clk(clk));

assign start_attack_pixel = snake_pixel | start_shoot;
assign mid_attack_pixel = snake_pixel | mid_shoot;
assign strike_attack_pixel = snake_pixel | strike_shoot;

always @(posedge clk) begin
  if (rst) begin
    state <= jaffar;
    timer_flag <= 1;
  end

  if (pixel_on) begin
    //timer for falling into the screen
    if (timer_flag) begin
      timer_flag <= 0;
      start_timer <= 1;
      timer_value <= 1;
    end
    //move on to next pixel state
    else if (timer_expired && state == jaffar) begin
      state <= snake;
      start_timer <= 1;
      timer_value <= 1;
    end
    else if (timer_expired && state == snake) begin
      state <= start_attack;
      start_timer <= 1;
      timer_value <= 1;
    end
    else if (timer_expired && state == start_attack) begin
      state <= strike_attack;
      start_timer <= 1;
      timer_value <= 2;
    end
    else if (timer_expired && state == mid_attack) begin
      state <= strike_attack;
      start_timer <= 1;
      timer_value <= 2;
    end
    else if (timer_expired && state == strike_attack) begin
      state <= start_attack;
      start_timer <= 1;
      timer_value <= 2;
    end else start_timer <= 0;

    case (state)
    snake: pixel <= snake_pixel;
    jaffar: pixel <= jaffar_pixel;
    start_attack: pixel <= start_attack_pixel;
    mid_attack: pixel <= mid_attack_pixel;
    strike_attack: pixel <= strike_attack_pixel;
    endcase

  end else begin
    state <= jaffar;
    timer_flag <= 1;
  end
end


endmodule

reg [10:0] y;
gen_rock rockblob (.x(x),.y(y),.hcount(hcount),.vcount(vcount),
                    .pixel_clk(clk), .pixel(rocks_pixel));

wire [2:0] hanging, falling, hitting;
assign hanging = 3'd1;
assign falling = 3'd2;
assign hitting = 3'b111;

reg fall_flag;

always @(posedge clk) begin
  if (level_state != 0 & game_level == 2'd2) begin
      if (pixel_on) begin
        if (!saved & fall_flag) begin
          if (state == hanging) begin
            state <= falling;
            start_timer <= 1;
            timer_value <= 1;
          end else if (state == falling & timer_expired) begin
            state <= hitting;
            start_timer <= 1;
            timer_value <= 1;
          end else if (state == hitting & timer_expired) begin
            state <= hanging;
            fall_flag <= 0;
          end else start_timer <= 0;
        end else state <= hanging;
      end else begin
        state <= hanging;
        fall_flag <= 1;
      end
    pixel <= rocks_pixel;
    case(state)
      hanging: y <= 210;
      falling: y <= 290;
      hitting: y <= 350;
    endcase
  end else state <= hanging;
  end

endmodule


module flashing(
    input [11:0] pixel,
    input clk,
    input flash,
    input timer_expired,
    output reg start_timer,
    output reg [6:0] timer_value,
    output reg [11:0] flashing_pixel
    );

    reg start_flashing, state;
    wire pixel_in, out;
    assign out = 0;
    assign pixel_in = 1;

    //maybe need seperate flash timer that counts less than a second?
    always @(posedge clk) begin
        if (flash) begin
            if (state == pixel_in && (start_flashing | timer_expired)) begin
                state <= out;
                start_timer <= 1;
                timer_value <= 1;
                start_flashing <= 0;
            end else if (state == out && timer_expired) begin
                state <= pixel_in;
                start_timer <= 1;
                timer_value <= 1;
            end else start_timer <= 0;
        end else begin
            state <= pixel_in;
            start_flashing <= 1;
        end

        case(state)
        out:         flashing_pixel <= 0;
        pixel_in:  flashing_pixel <= pixel;
        endcase
    end
endmodule


`timescale 1ns / 1ps

module Flash_Timer(
    input clk,
    input start_timer,
    input [6:0] value,
    output reg expired,
    output reg [6:0] counter
    );

    //create the seconds counter
    wire count_enable;
    half_second_dividor quarter_sec_counter(.clk(clk),
.timer_enable(start_timer), .one(count_enable));

    //reg [6:0] counter;
    reg flag;

    always @ (posedge clk) begin
    //if start_timer is asserted, initialize the variables
    if (start_timer) begin
        if (value == 0) begin
            expired <= 1;
        end

        else begin
            counter <= value;
            expired <= 0;
            flag <= 1;
         end
    end

    //"else if" to give priority to start_timer if needs to restart
    //decrement the counter at each second enabled
    else if (count_enable && counter > 0) begin
        counter <= counter - 1;
    end

    //if the counter has reached the requested value, assert expired
    else if ((counter == 0) && flag && !expired) begin
        expired <= 1;
        flag <= 0;
    end

    else if (expired && !flag) expired <= 0;

    end
endmodule


module Villain_Selector(
    input clk,
    input rst,
    input ongoing,
    input [2:0] level_state,
    input [1:0] game_level,
    output reg [2:0] villain
    );


    //define the level states of each level
    wire [2:0] idle, start_level, obstacle_1, rest_1, obstacle_2, rest_2,
obstacle_3, rest_3;
    assign idle = 3'b0;
    assign start_level = 3'b1;
    assign obstacle_1 = 3'd2;
    assign rest_1 = 3'd3;
    assign obstacle_2 = 3'd4;
    assign rest_2 = 3'd5;
    assign obstacle_3 = 3'd6;
    assign rest_2 = 3'd7;

    //define each villain
    wire [2:0] rocks, lava, imperial_guard, snake;
    assign rocks = 3'd1;
    assign lava = 3'd2;
    assign imperial_guard = 3'd3;
    assign snake = 3'd4;

always @(posedge clk) begin
    if (rst) villain <=3'd0;
    if (ongoing & (level_state == obstacle_1 | level_state == obstacle_2 |
level_state == obstacle_3)) begin
      case(game_level)
      3'd1: villain <= imperial_guard;
      3'd2: villain <= rocks;
      3'd3: villain <= snake;
     endcase
    end else villain <= 0;
end

endmodule

module Imperial_Guard(
    input clk,
    input rst,
    input pixel_on,
    input timer_expired,
    input [9:0] hcount,
    input [10:0] vcount,
    output reg start_timer,
    output reg [6:0] timer_value,
    output reg [2:0] state,
    output reg [11:0] pixel
    );

//define the states of the guard
wire [2:0] fall_in, stand_there, combat;
assign fall_in = 3'd1;
assign stand_there = 3'd2;
assign combat = 3'b111;

reg timer_flag;

//define blobs for each state
wire [11:0] fall_in_pixel, stand_there_pixel, combat_pixel;

badguy_mux fallblob(.x(370),.y(200),.hcount(hcount),.vcount(vcount),
                      .im_sel(3'd0), .pixel(fall_in_pixel), .clk(clk));

badguy_mux standblob(.x(370),.y(350),.hcount(hcount),.vcount(vcount),
                      .im_sel(3'd0), .pixel(stand_there_pixel), .clk(clk));

badguy_mux combatblob(.x(370),.y(350),.hcount(hcount),.vcount(vcount),
                      .im_sel(3'd1), .pixel(combat_pixel), .clk(clk));

always @(posedge clk) begin
  if (rst) begin
```

```
      state <= fall_in;
      timer_flag <= 1;
    end

    if (pixel_on) begin
      //timer for falling into the screen
      if (timer_flag) begin
        timer_flag <= 0;
        start_timer <= 1;
        timer_value <= 1; //falling in for 1 sec
      end
      //move on to next pixel state
      else if (timer_expired && state == fall_in) begin
        state <= stand_there;
        start_timer <= 1;
        timer_value <= 1; //standing for 1 sec
      end
      else if (timer_expired && state == stand_there) begin
        state <= combat;
        start_timer <= 1;
        timer_value <= 1; //combating for 1 sec
      end
      else if (timer_expired && state == combat) begin
        state <= stand_there;
        start_timer <= 1;
        timer_value <= 2; //standing for 2 sec
      end else start_timer <= 0;

      case (state)
      fall_in: pixel <= fall_in_pixel;
      stand_there: pixel <= stand_there_pixel;
      combat: pixel <= combat_pixel;
      endcase
    end else begin
      state <= fall_in;
      timer_flag <= 1;
    end
  end

  endmodule


  `timescale 1ns / 1ps

  module Score_Unit(
      input clk,
      input rst,
      input ongoing,
      input [1:0] game_level,
      input good_collision,
      input bad_collision,
      output reg signed [11:0] score = 0
      );

  wire [10:0] penalty_1, reward_1, penalty_2, reward_2, penalty_3, reward_3;
  assign penalty_1 = 10'd10;
  assign reward_1 = 10'd30;
  assign penalty_2 = 10'd20;
  assign reward_2 = 10'd60;
  assign penalty_3 = 10'd40;
  assign reward_3 = 10'd120;

  reg count_score_flag = 1;

  always @(posedge clk) begin
      //initialize the score of the game
      if (rst) begin
          score <= 0;
          count_score_flag <= 1;
      end

      if (good_collision | bad_collision) count_score_flag <= 0;
      if (!good_collision && !bad_collision) count_score_flag <= 1;

      if (ongoing & count_score_flag) begin
          case (game_level)
          2'd1: begin
                 if (good_collision) score <= (score+reward_1 >
  12'b011111111111)? 12'b011111111111: score + reward_1;
                 else if (bad_collision)
                      score <= (score < penalty_1)? 0 : score - penalty_1;
                end
          2'd2: begin
                 if (good_collision) score <= (score+reward_2 >
  12'b011111111111)? 12'b011111111111: score + reward_2;
                 else if (bad_collision) score <= (score < penalty_2)? 0 : score
  - penalty_2;
               end
          2'd3: begin
                 if (good_collision) score <= (score+reward_3 >
  12'b011111111111)? 12'b011111111111: score + reward_3;
                 else if (bad_collision) score <= (score < penalty_3)? 0 : score
  - penalty_3;
                end
          endcase
      end else if (!ongoing) begin
          count_score_flag <= 1;
      end

  end
  endmodule

  `timescale 1ns / 1ps

  module score_graphics(
      input clk,
      input [10:0] score,
      input [2:0] level_state,
      input [9:0] hcount,
      input [10:0] vcount,
      input [2:0] game_level,
      output reg [11:0] pixel
      );

      wire [2:0] idle;
      assign idle = 3'b0;

      parameter thickness = 2;
      parameter width = 25;
      parameter height = 35;
      parameter space = 10;
      parameter pos = 50;


  //////////////////////////////////////////////////////////////////////////////
  ///////////////////
      //                                          LETTERS

      wire [11:0] s1,s2,s3,s4,s5,s, c1,c2,c3,c, o1,o2,o3,o4,o, r1,r2,r3,r4,r5,r,
  e1,e2,e3,e4,e;
      parameter color = 12'hFFF;
      blob #(.WIDTH(width),.HEIGHT(thickness),.COLOR(color)) // yellow
         s1blob(.x(pos),.y(pos),.hcount(hcount),.vcount(vcount),
                .pixel(s1));
      blob #(.WIDTH(thickness),.HEIGHT(height/2),.COLOR(color)) // yellow
         s2blob(.x(pos),.y(pos),.hcount(hcount),.vcount(vcount),
                .pixel(s2));
      blob #(.WIDTH(width),.HEIGHT(thickness),.COLOR(color)) // yellow

  s3blob(.x(pos),.y(pos+height/2-thickness),.hcount(hcount),.vcount(vcount),
                  .pixel(s3));
      blob #(.WIDTH(thickness),.HEIGHT(height/2),.COLOR(color)) // yellow
```

```
s4blob(.x(pos+width-thickness),.y(pos+height/2/*-thickness*/),.hcount(hcount),
.vcount(vcount),
               .pixel(s4));
    blob #(.WIDTH(width),.HEIGHT(thickness),.COLOR(color)) // yellow

s5blob(.x(pos),.y(pos+height-thickness),.hcount(hcount),.vcount(vcount),
               .pixel(s5));
    assign s = s1 | s2 | s3 | s4 | s5;

    parameter c_pos = pos+width+space;
    blob #(.WIDTH(width),.HEIGHT(thickness),.COLOR(color)) // yellow
       c1blob(.x(c_pos),.y(pos),.hcount(hcount),.vcount(vcount),
               .pixel(c1));
    blob #(.WIDTH(thickness),.HEIGHT(height),.COLOR(color)) // yellow
       c2blob(.x(c_pos),.y(pos),.hcount(hcount),.vcount(vcount),
               .pixel(c2));
    blob #(.WIDTH(width),.HEIGHT(thickness),.COLOR(color)) // yellow

c3blob(.x(c_pos),.y(pos+height-thickness),.hcount(hcount),.vcount(vcount),
               .pixel(c3));
    assign c = c1 | c2 | c3;

    parameter o_pos = pos+2*width+2*space;
     blob #(.WIDTH(width),.HEIGHT(thickness),.COLOR(color)) // yellow
        o1blob(.x(o_pos),.y(pos),.hcount(hcount),.vcount(vcount),
                .pixel(o1));
     blob #(.WIDTH(thickness),.HEIGHT(height),.COLOR(color)) // yellow
        o2blob(.x(o_pos),.y(pos),.hcount(hcount),.vcount(vcount),
                .pixel(o2));
     blob #(.WIDTH(width),.HEIGHT(thickness),.COLOR(color)) // yellow

o3blob(.x(o_pos),.y(pos+height-thickness),.hcount(hcount),.vcount(vcount),
                .pixel(o3));
    blob #(.WIDTH(thickness),.HEIGHT(height),.COLOR(color)) // yellow

o4blob(.x(o_pos+width-thickness),.y(pos),.hcount(hcount),.vcount(vcount),
               .pixel(o4));
    assign o = o1 | o2 | o3 | o4;

     parameter r_pos = pos+3*width+3*space;

     blob #(.WIDTH(width),.HEIGHT(thickness),.COLOR(color)) // yellow
        r1blob(.x(r_pos),.y(pos),.hcount(hcount),.vcount(vcount),
                .pixel(r1));
     blob #(.WIDTH(thickness),.HEIGHT(height),.COLOR(color)) // yellow
        r2blob(.x(r_pos),.y(pos),.hcount(hcount),.vcount(vcount),
                .pixel(r2));
     blob #(.WIDTH(width),.HEIGHT(thickness),.COLOR(color)) // yellow

r3blob(.x(r_pos),.y(pos+height/2-thickness),.hcount(hcount),.vcount(vcount),
                .pixel(r3));
     blob #(.WIDTH(thickness),.HEIGHT(height/2),.COLOR(color)) // yellow

r4blob(.x(r_pos+width-thickness),.y(pos),.hcount(hcount),.vcount(vcount),
                .pixel(r4));
     blob #(.WIDTH(thickness),.HEIGHT(height/2),.COLOR(color)) // yellow

r5blob(.x(r_pos+width*3/4),.y(pos+height/2),.hcount(hcount),.vcount(vcount),
                .pixel(r5));
    assign r = r1 | r2 | r3 | r4 | r5;

    parameter e_pos = pos+4*width+4*space;
     blob #(.WIDTH(thickness),.HEIGHT(height),.COLOR(color)) // yellow
        e1blob(.x(e_pos),.y(pos),.hcount(hcount),.vcount(vcount),
                .pixel(e1));
     blob #(.WIDTH(width),.HEIGHT(thickness),.COLOR(color)) // yellow
        e2blob(.x(e_pos),.y(pos),.hcount(hcount),.vcount(vcount),
                .pixel(e2));
     blob #(.WIDTH(width),.HEIGHT(thickness),.COLOR(color)) // yellow

e3blob(.x(e_pos),.y(pos+height/2-thickness),.hcount(hcount),.vcount(vcount),
                .pixel(e3));
    blob #(.WIDTH(width),.HEIGHT(thickness),.COLOR(color)) // yellow

e4blob(.x(e_pos),.y(pos+height-thickness),.hcount(hcount),.vcount(vcount),
               .pixel(e4));
    assign e = e1 | e2 | e3 | e4;


///////////////////////////////////////////////////////////////////////////////
////////////////////
    //                                           NUMBERS

    parameter pos_1 = e_pos + 2*space + 1*width;     parameter pos_2 = e_pos +
3*space + 2*width;
    parameter pos_3 = e_pos + 4*space + 3*width;     parameter pos_4 = e_pos +
5*space + 4*width;
    parameter pos_5 = e_pos + 6*space + 5*width;     parameter pos_6 = e_pos +
7*space + 6*width;
    parameter pos_7 = e_pos + 8*space + 7*width;     parameter pos_8 = e_pos +
9*space + 8*width;
    parameter pos_9 = e_pos + 10*space + 9*width;    parameter pos_10 = e_pos
+ 11*space + 10*width;
    parameter pos_11 = e_pos + 12*space + 11*width;
    wire [11:0] one_1, one_2, one_3, one_4, one_5, one_6, one_7, one_8, one_9,
one_10, one_11;
    wire [11:0] zero_1, zero_2, zero_3, zero_4, zero_5, zero_6, zero_7,
zero_8, zero_9, zero_10, zero_11;

    one one1(.x(pos_1), .y(pos), .hcount(hcount), .vcount(vcount),
.pixel(one_1));
    one one2(.x(pos_2), .y(pos), .hcount(hcount), .vcount(vcount),
.pixel(one_2));
    one one3(.x(pos_3), .y(pos), .hcount(hcount), .vcount(vcount),
.pixel(one_3));
    one one4(.x(pos_4), .y(pos), .hcount(hcount), .vcount(vcount),
.pixel(one_4));
    one one5(.x(pos_5), .y(pos), .hcount(hcount), .vcount(vcount),
.pixel(one_5));
    one one6(.x(pos_6), .y(pos), .hcount(hcount), .vcount(vcount),
.pixel(one_6));
    one one7(.x(pos_7), .y(pos), .hcount(hcount), .vcount(vcount),
.pixel(one_7));
    one one8(.x(pos_8), .y(pos), .hcount(hcount), .vcount(vcount),
.pixel(one_8));
    one one9(.x(pos_9), .y(pos), .hcount(hcount), .vcount(vcount),
.pixel(one_9));
    one one10(.x(pos_10), .y(pos), .hcount(hcount), .vcount(vcount),
.pixel(one_10));
    one one11(.x(pos_11), .y(pos), .hcount(hcount), .vcount(vcount),
.pixel(one_11));

    zero zero1(.x(pos_1), .y(pos), .hcount(hcount), .vcount(vcount),
.pixel(zero_1));
    zero zero2(.x(pos_2), .y(pos), .hcount(hcount), .vcount(vcount),
.pixel(zero_2));
    zero zero3(.x(pos_3), .y(pos), .hcount(hcount), .vcount(vcount),
.pixel(zero_3));
    zero zero4(.x(pos_4), .y(pos), .hcount(hcount), .vcount(vcount),
.pixel(zero_4));
    zero zero5(.x(pos_5), .y(pos), .hcount(hcount), .vcount(vcount),
.pixel(zero_5));
    zero zero6(.x(pos_6), .y(pos), .hcount(hcount), .vcount(vcount),
.pixel(zero_6));
    zero zero7(.x(pos_7), .y(pos), .hcount(hcount), .vcount(vcount),
.pixel(zero_7));
    zero zero8(.x(pos_8), .y(pos), .hcount(hcount), .vcount(vcount),
.pixel(zero_8));
    zero zero9(.x(pos_9), .y(pos), .hcount(hcount), .vcount(vcount),
.pixel(zero_9));
    zero zero10(.x(pos_10), .y(pos), .hcount(hcount), .vcount(vcount),
.pixel(zero_10));
```

```
    zero zero11(.x(pos_11), .y(pos), .hcount(hcount), .vcount(vcount),
.pixel(zero_11));

    reg [11:0] score_value;
always @(posedge clk) begin
    casex(score)
    11'b1XXXXXXXXXX:  score_value <= one_1 | (score[9]? one_2: zero_2) |
(score[8]? one_3: zero_3) |
                                        (score[7]?one_4:zero_4) |
(score[6]?one_5:zero_5) |
                                        (score[5]?one_6:zero_6) |
(score[4]?one_7:zero_7) |
                                        (score[3]?one_8:zero_8) |
(score[2]?one_9:zero_9) |
                                        (score[1]?one_10:zero_10) |
(score[0]?one_11:zero_11) ;
    11'b01XXXXXXXXX: score_value <= one_1 | (score[8]? one_2: zero_2) |
(score[7]? one_3: zero_3) |
                                        (score[6]?one_4:zero_4) |
(score[5]?one_5:zero_5) |
                                        (score[4]?one_6:zero_6) |
(score[3]?one_7:zero_7) |
                                        (score[2]?one_8:zero_8) |
(score[1]?one_9:zero_9) |
                                        (score[0]?one_10:zero_10);
    11'b001XXXXXXXX: score_value <= one_1 | (score[7]? one_2: zero_2) |
(score[6]? one_3: zero_3) |
                                        (score[5]?one_4:zero_4) |
(score[4]?one_5:zero_5) |
                                        (score[3]?one_6:zero_6) |
(score[2]?one_7:zero_7) |
                                        (score[1]?one_8:zero_8) |
(score[0]?one_9:zero_9);
    11'b0001XXXXXXX: score_value <= one_1 | (score[6]? one_2: zero_2) |
(score[5]? one_3: zero_3) |
                                        (score[4]?one_4:zero_4) |
(score[3]?one_5:zero_5) |
                                        (score[2]?one_6:zero_6) |
(score[1]?one_7:zero_7) |
                                        (score[0]?one_8:zero_8);
    11'b00001XXXXXX: score_value <= one_1 | (score[5]? one_2: zero_2) |
(score[4]? one_3: zero_3) |
                                        (score[3]?one_4:zero_4) |
(score[2]?one_5:zero_5) |
                                        (score[1]?one_6:zero_6) |
(score[0]?one_7:zero_7);
    11'b000001XXXXX: score_value <= one_1 | (score[4]? one_2: zero_2) |
(score[3]? one_3: zero_3) |
                                        (score[2]?one_4:zero_4) |
(score[1]?one_5:zero_5) |
                                        (score[0]?one_6:zero_6);
    11'b0000001XXXX: score_value <= one_1 | (score[3]? one_2: zero_2) |
(score[2]? one_3: zero_3) |
                                        (score[1]?one_4:zero_4) |
(score[0]?one_5:zero_5);
    11'b00000001XXX: score_value <= one_1 | (score[2]? one_2: zero_2) |
(score[1]? one_3: zero_3) |
                                      (score[0]?one_4:zero_4);
    11'b000000001XX: score_value <= one_1 | (score[1]? one_2: zero_2) |
(score[0]? one_3: zero_3);
    11'b0000000001X: score_value <= one_1 | (score[0]? one_2: zero_2);
    default: score_value <= score[0]? one_1 : zero_1;
    endcase

    if (level_state != idle | game_level == 3'd4)
        pixel <= s | c | o | r | e | score_value;
    else pixel <= 0;
end
endmodule

`timescale 1ns / 1ps


module level_cover(
    input clk,
    input [9:0] hcount,
    input [10:0] vcount,
    input [2:0] level_state,
    input [2:0] level,
    output reg [11:0] pixel
    );

    wire [2:0] idle;
    assign idle = 3'd0;

    wire[11:0] l1, l2, l, zero1, zero2, one1, one2, n1, n2, n3;
    parameter height = 200;
    parameter width = 70;
    parameter thickness = 20;
    parameter x_pos = 200;
    parameter y_pos = 150;
    parameter color = 12'hEEE;

    blob #(.WIDTH(thickness),.HEIGHT(height),.COLOR(color)) // white
       l1blob(.x(x_pos),.y(y_pos),.hcount(hcount),.vcount(vcount),
              .pixel(l1));
    blob #(.WIDTH(width),.HEIGHT(thickness),.COLOR(color)) // white

l2blob(.x(x_pos),.y(y_pos+height-thickness),.hcount(hcount),.vcount(vcount),
                .pixel(l2));
    assign l = l1 | l2;

    zero #(.WIDTH(width), .HEIGHT(height), .THICKNESS(thickness),
.COLOR(color))
         zero_1(.x(300), .hcount(hcount), .y(y_pos), .vcount(vcount),
.pixel(zero1));

    one #(.WIDTH(width), .HEIGHT(height), .THICKNESS(thickness),
.COLOR(color))
         one_1(.x(300), .hcount(hcount), .y(y_pos), .vcount(vcount),
.pixel(one1));

    zero #(.WIDTH(width), .HEIGHT(height), .THICKNESS(thickness),
.COLOR(color))
          zero_2(.x(400), .hcount(hcount), .y(y_pos), .vcount(vcount),
.pixel(zero2));

    one #(.WIDTH(width), .HEIGHT(height), .THICKNESS(thickness),
.COLOR(color))
          one_2(.x(400), .hcount(hcount), .y(y_pos), .vcount(vcount),
.pixel(one2));

    assign n1 = zero1 | one2;
    assign n2 = one1 | zero2;
    assign n3 = one1 | one2;

    wire [11:0] kwala_pixel, flash_press_start, press_start, game_over;
    reg [10:0] y = 0;
    gen_koala k(.x(100),.y(y),.hcount(hcount),.vcount(vcount),
                .pixel(kwala_pixel), .pixel_clk(clk));
    gen_start s(.x(10),.y(410),.hcount(hcount),.vcount(vcount),
                            .pixel(press_start), .pixel_clk(clk));
    gen_gameover g(.x(40),.y(410),.hcount(hcount),.vcount(vcount),
                    .pixel(game_over), .pixel_clk(clk));

    flashing flash(.pixel(press_start), .clk(clk), .flash(1),
.timer_expired(flash_timer_expired),
                   .start_timer(flash_start_timer),
.timer_value(flash_timer_value),
                   .flashing_pixel(flash_press_start));
    Flash_Timer flashTimer(.clk(clk), .start_timer(flash_start_timer),
.value(flash_timer_value), .expired(flash_timer_expired),
                           .counter(flash_counter));
```

```
always @(posedge clk) begin
    if (level_state == idle) begin
        case (level)
        3'd0: begin
            y <= 0;
            pixel <= kwala_pixel | flash_press_start;
            end
        3'd1: pixel <= ~(l | n1);
        3'd2: pixel <= ~(l | n2);
        3'd3: pixel <= ~(l | n3);
        3'd4: begin
            y <= 80;
            pixel <= (game_over != 12'h000)? game_over: kwala_pixel;
            end
        endcase
    end else pixel <= 0;
end

endmodule




module one#(parameter WIDTH = 25,            // default width: 25 pixels
                HEIGHT = 35,           // default height: 35 pixels
                THICKNESS = 2,
                COLOR = 12'hFF0)  // default color: yellow
    (input [10:0] x,hcount,
     input [9:0] y,vcount,
     output reg [11:0] pixel);

     wire [11:0] one;
     blob #(.WIDTH(THICKNESS),.HEIGHT(HEIGHT),.COLOR(COLOR)) // yellow
       o4blob(.x(x+WIDTH-THICKNESS),.y(y),.hcount(hcount),.vcount(vcount),
               .pixel(one));

always @ *
        pixel <= one;

endmodule

`timescale 1ns / 1ps

module zero#(parameter WIDTH = 25,            // default width: 25 pixels
                HEIGHT = 35,           // default height: 35 pixels
                THICKNESS = 2,
                COLOR = 12'hFF0)  // default color: yellow
    (input [10:0] x,hcount,
     input [9:0] y,vcount,
     output reg [11:0] pixel);

     wire [11:0] o1, o2, o3, o4;
     blob #(.WIDTH(WIDTH),.HEIGHT(THICKNESS),.COLOR(COLOR)) // yellow
        o1blob(.x(x),.y(y),.hcount(hcount),.vcount(vcount),
                .pixel(o1));
     blob #(.WIDTH(THICKNESS),.HEIGHT(HEIGHT),.COLOR(COLOR)) // yellow
        o2blob(.x(x),.y(y),.hcount(hcount),.vcount(vcount),
                .pixel(o2));
     blob #(.WIDTH(WIDTH),.HEIGHT(THICKNESS),.COLOR(COLOR)) // yellow
        o3blob(.x(x),.y(y+HEIGHT-THICKNESS),.hcount(hcount),.vcount(vcount),
                .pixel(o3));
    blob #(.WIDTH(THICKNESS),.HEIGHT(HEIGHT),.COLOR(COLOR)) // yellow
       o4blob(.x(x+WIDTH-THICKNESS),.y(y),.hcount(hcount),.vcount(vcount),
               .pixel(o4));

always @ *
        pixel <= o1| o2| o3| o4;

endmodule

`timescale 1ns / 1ps

module sound_effects(
    input clk,
    input good_collide,
    input bad_collide,
    output reg frequency
    );

reg [15:0] count;
reg [15:0] high_freq_threshold;
reg [15:0] low_freq_threshold;

always @(posedge clk) begin
    if (good_collide) begin
        if (count == high_freq_threshold) begin
           frequency <= !frequency;
           count <= 0;
           high_freq_threshold <= high_freq_threshold + 200;
        end else    count <= 1 + count;
    end else if (bad_collide) begin
        if (count == high_freq_threshold) begin
            frequency <= !frequency;
            count <= 0;
            low_freq_threshold <= low_freq_threshold - 200;
        end else    count <= 1 + count;
    end else begin
        count <= 0;
        frequency <= 0;
        high_freq_threshold <= 25000;
        low_freq_threshold <= 35000;
    end
    end
endmodule


`timescale 1ns / 1ps


module NES_controller(
    input clk_25mhz,
    input data,
    output reg latch,
    output reg pulse,
    output reg A, B, select, start, up, down, left, right,
    output reg latch_sent=0, data_finalized=0,
    output reg [3:0]  pulse_received=0,
    output reg [3:0] input_count
    );

    wire clk_60hz, timer_expired_12us, timer_expired_6us;
    reg  start_timer_12us, start_timer_6us;
    gen_clk_60hz new_clk(.clk_25mhz(clk_25mhz), .clk_60hz(clk_60hz));
    timer_12us timer_12(.clk_25mhz(clk_25mhz), .start_timer(start_timer_12us),
.timer_expired(timer_expired_12us));
    timer_6us timer_6(.clk_25mhz(clk_25mhz), .start_timer(start_timer_6us),
.timer_expired(timer_expired_6us));

    reg [7:0] controller_in;
    //reg [3:0] input_count;
    reg [3:0] pulse_count;
    reg sample_data_flag;

    always @ (posedge clk_25mhz) begin
        //send a latch signal for 12us at 60Hz
        if (clk_60hz) begin
            start_timer_12us <= 1;
            latch <= 1;
            input_count <= 0;
            pulse_count <= 0;
            pulse <= 0;
            sample_data_flag <= 0;
            controller_in <= 0;
            latch_sent <= 1;
        end else if (timer_expired_12us) begin
            latch <= 0;
            start_timer_6us <= 1;
            sample_data_flag <= 1;
        end else begin
            start_timer_12us <= 0;
            start_timer_6us <= 0;
        end

        //after the latch signal, listen in for data on pulse
        if (input_count == 4'd8) begin
            {A, B, select, start, up, down, left, right} <= ~controller_in;
//negative true
            input_count <= 0;
            data_finalized <= 1;
        end else if (timer_expired_6us & pulse_count <= 4'd7) begin
            pulse <= !pulse;
            start_timer_6us <= 1;
            pulse_count <= pulse_count + pulse;
            sample_data_flag <= pulse; // set sample flag on the falling edge
        end else if (sample_data_flag & input_count < 4'd8) begin
            pulse <= 0;
            controller_in <= {controller_in[6:0], data};
            input_count <= input_count + 1;
            pulse_received <= pulse_received+1;
            sample_data_flag <= 0;
        end

    end
endmodule

`timescale 1ns / 1ps

module timer_6us(
    input clk_25mhz,
input start_timer,
output reg timer_expired
);

reg [8:0] count = 0;
reg  counting;

always @ (posedge clk_25mhz) begin
    if (count == 9'd150) begin
        timer_expired <= 1;
        counting <= 0;
        count <= 0;
    end else if (counting) begin
```

```
        count <= count + 1;
    end else if (start_timer) begin
        counting <= 1;
        timer_expired <= 0;
        //count <= 0;
    end else timer_expired <= 0;
 end
endmodule

`timescale 1ns / 1ps


module timer_12us(
    input clk_25mhz,
    input start_timer,
    output reg timer_expired
    );

    reg [8:0] count = 0;
    reg  counting;

    always @ (posedge clk_25mhz) begin
        if (count == 9'd300) begin
            timer_expired <= 1;
            counting <= 0;
            count <= 0;
        end else if (counting) begin
            count <= count + 1;
        end else if (start_timer) begin
            counting <= 1;
            timer_expired <= 0;
            //count <= 0;
        end else timer_expired <= 0;
     end
endmodule

`timescale 1ns / 1ps

module gen_clk_60hz(
    input clk_25mhz,
    output reg clk_60hz = 0
    );
    reg [18:0]  count = 0;
    always @(posedge clk_25mhz) begin
        if (count == 19'd416667) begin
            clk_60hz <= 1;
            count <= 0;
        end else begin
```

```
            count <= count + 1;
            clk_60hz <= 0;
        end
    end
endmodule


`timescale 1ns / 1ps

module labkit(
   input CLK100MHZ,
   input[15:0] SW,
   input BTNC, BTNU, BTNL, BTNR, BTND,
   output[3:0] VGA_R,
   output[3:0] VGA_B,
   output[3:0] VGA_G,
   inout[7:0] JA,
   //input [7:0] JA,
   output VGA_HS,
   output VGA_VS,
   output LED16_B, LED16_G, LED16_R,
   output LED17_B, LED17_G, LED17_R,
   output[15:0] LED,
   output[7:0] SEG,  // segments A-G (0-6), DP (7)
   output[7:0] AN    // Display 0-7
   );

// create 25mhz system clock
    wire clock_25mhz;
    clock_quarter_divider clockgen(.clk100_mhz(CLK100MHZ),
.clock_25mhz(clock_25mhz));

//  instantiate 7-segment display;
    wire [31:0] data;
    wire [6:0] segments;
    display_8hex display(.clk(clock_25mhz),.data(data), .seg(segments),
.strobe(AN));
    assign SEG[6:0] = segments;
    assign SEG[7] = 1'b1;


////////////////////////////////////////////////////////////////////////////////
//////////////

////////////////////////////////////////////////////////////////////////////////
////////////
```

```
////////////////////////////////////////////////////////////////////////////////
//////////
    //Integrate all modules here:


////////////////////////////////////////////////////////////////////////////////
//////////////
    // NES CONTROLLER INPUT
    wire A, B, select, nstart, up, down, nleft, nright;
    wire [3:0] pulse_received, input_count, data_finalized;
    wire latch_sent;
    NES_controller controller(.clk_25mhz(clock_25mhz), .pulse(JA[3]),
.data(JA[2]),
                              .latch(JA[1]), .A(A), .B(B), .select(select),
.start(nstart),
                              .up(up), .down(down), .left(nleft),
.right(nright),
                              .latch_sent(latch_sent),
.pulse_received(pulse_received), .data_finalized(data_finalized),
.input_count(input_count));
    assign LED[7:0] = {A, B, select, nstart, up, down, nleft, nright};


////////////////////////////////////////////////////////////////////////////////
////////////
    //debounce all the inputs:
    wire reset, fleft, fright, fpick_carpet, fstart;
    assign reset = SW[15];
    //debounce debounce_reprogram(.reset(SW[15]), .clock(clock_25mhz),
.noisy(BTNC), .clean(reset));
    debounce debounce_left(.reset(reset), .clock(clock_25mhz), .noisy(BTNL),
.clean(fleft));
    debounce debounce_right(.reset(reset), .clock(clock_25mhz), .noisy(BTNR),
.clean(fright));
    debounce debounce_pick(.reset(reset), .clock(clock_25mhz), .noisy(BTNU),
.clean(fpick_carpet));
    debounce debounce_start(.reset(reset), .clock(clock_25mhz), .noisy(BTNC),
.clean(fstart));

    wire left, right, pick_carpet, start;
    assign left = fleft | nleft;
    assign right = fright | nright;
    assign start = fstart | nstart;
    assign pick_carpet = fpick_carpet | A;

    wire [9:0] hcount;
    wire [10:0] vcount;
    wire hsync, vsync, at_display_area, blank;
    //create the vga display
    vga vga1(.vga_clock(clock_25mhz),.hcount(hcount),.vcount(vcount),
          .hsync(hsync),.vsync(vsync),.at_display_area(at_display_area));

    assign VGA_HS = ~hsync;
    assign VGA_VS = ~vsync;


////////////////////////////////////////////////////////////////////////////////
////////////////////////////
    //instantiate game FSM
    wire game_start_timer, game_timer_expired, ongoing;
    wire [6:0] game_timer_value, game_counter;
    wire [2:0] game_level;
    wire [2:0] level_state;
    Game_FSM gamefsm(.clk(clock_25mhz), .rst(reset),
/*.bad_collision(bad_collide),*/ .timer_expired(game_timer_expired),
                      .start_timer(game_start_timer),
.timer_value(game_timer_value), .ongoing(ongoing), .level_state(level_state),
.game_level(game_level),
                      /*.player_item(player_item),*/ .start(start));
    Timer game_FSM_timer(.clk(clock_25mhz), .start_timer(game_start_timer),
.value(game_timer_value), .expired(game_timer_expired),
                         .counter(game_counter));

    wire player_position, player_start_timer, player_timer_expired,
timer_flag, apple_throw;
    wire [11:0] player_pixel;
    wire [6:0] player_timer_value, player_counter;
    //create the player and its own timer
    Timer player_timer(.clk(clock_25mhz), .start_timer(player_start_timer),
.value(player_timer_value), .expired(player_timer_expired),
                        .counter(player_counter));
    Player_FSM player(.clk(clock_25mhz), .ongoing(ongoing), .rst(reset),
.left(left), .right(right), .timer_expired(player_timer_expired),
                        .start_timer(player_start_timer),
.timer_value(player_timer_value), .player_position(player_position),
                        .hcount(hcount), .vcount(vcount),
.pixel(player_pixel), .timer_flag(timer_flag), .vsync(vsync),
.pick_carpet(pick_carpet),
                        .level_state(level_state), .game_level(game_level),
.apple_throw(apple_throw));


////////////////////////////////////////////////////////////////////////////////
/////////////////////////////
    //instantiate item selector module
    wire [2:0] item;
    //define each item
    wire [2:0] apples, carpet, sword;
    assign apples = 3'd1;
    assign carpet = 3'd2;
    assign sword = 3'd3;

    item_selector is(.clk(clock_25mhz), .rst(reset), .ongoing(ongoing),
.level_state(level_state), .game_level(game_level), .item(item));

    wire apples_on, carpet_on, sword_on;
    assign apples_on = item == apples;
    assign carpet_on = item == carpet;
    assign sword_on =  item == sword;

    //apples
    wire apples_start_timer, apples_timer_expired;
    wire [6:0] apples_timer_value, apples_counter;
    wire [2:0] apples_state;
    wire [11:0] apples_pixel;

    apples apple_item(.clk(clock_25mhz), .item_on(apples_on),
.player_position(player_position), .vcount(vcount), .hcount(hcount),
                      .start_timer(apples_start_timer),
.timer_value(apples_timer_value), .timer_expired(apples_timer_expired),
                      .state(apples_state), .pixel(apples_pixel),
.picked(good_collide));
    Timer apples_timer(.clk(clock_25mhz), .start_timer(apples_start_timer),
.value(apples_timer_value), .expired(apples_timer_expired),
                                          .counter(apples_counter));

    //sword
    wire sword_start_timer, sword_timer_expired;
    wire [6:0] sword_timer_value, sword_counter;
    wire [2:0] sword_state;
    wire [11:0] sword_pixel;

    sword sword_item(.clk(clock_25mhz), .item_on(sword_on),
.player_position(player_position), .vcount(vcount), .hcount(hcount),
                      .start_timer(sword_start_timer),
.timer_value(sword_timer_value), .timer_expired(sword_timer_expired),
                      .state(sword_state), .pixel(sword_pixel),
.picked(good_collide));
    Timer sword_timer(.clk(clock_25mhz), .start_timer(sword_start_timer),
.value(sword_timer_value), .expired(sword_timer_expired),
                                          .counter(sword_counter));

    //carpet
    wire carpet_start_timer, carpet_timer_expired;
    wire [6:0] carpet_timer_value, carpet_counter;
    wire [2:0] carpet_state;
    wire [11:0] carpet_pixel;

    Flying_Carpet carpet_item(.clk(clock_25mhz), .pick_carpet(pick_carpet),
.timer_expired(carpet_timer_expired), .game_level(game_level),
                          .level_state(level_state), .hcount(hcount),
.vcount(vcount), .start_timer(carpet_start_timer),
                          .timer_value(carpet_timer_value),
.pixel(carpet_pixel));
    Timer carpet_timer(.clk(clock_25mhz), .start_timer(carpet_start_timer),
.value(carpet_timer_value), .expired(carpet_timer_expired),

.counter(carpet_counter));

    //monkey
    wire monkey_start_timer, monkey_timer_expired, monkey_saved;
    wire [6:0] monkey_timer_value, monkey_counter;
    wire [2:0] monkey_state;
    wire [11:0] monkey_pixel;

    monkey monkey_item(.clk(clock_25mhz), .pick_carpet(pick_carpet),
.timer_expired(monkey_timer_expired), .game_level(game_level),
                          .level_state(level_state), .hcount(hcount),
.vcount(vcount), .start_timer(monkey_start_timer),
                          .timer_value(monkey_timer_value),
.pixel(monkey_pixel), .item_on(carpet_on), .saved(monkey_saved),
                          .state(monkey_state),
.player_position(player_position));
    Timer monkey_timer(.clk(clock_25mhz), .start_timer(monkey_start_timer),
.value(monkey_timer_value), .expired(monkey_timer_expired),

.counter(monkey_counter));


////////////////////////////////////////////////////////////////////////////////
////////////////////////////
    //instantiate villain selector module
    wire [2:0] villain;
    //define each villain
    wire [2:0] rocks, lava, imperial_guard, snake;
    assign rocks = 3'd1;
    assign lava = 3'd2;
    assign imperial_guard = 3'd3;
    assign snake = 3'd4;
    Villain_Selector vs(.clk(clock_25mhz), .rst(reset), .ongoing(ongoing),
.level_state(level_state), .game_level(game_level), .villain(villain));

    wire imperial_guard_on, rocks_on, lava_on, snake_on;
    assign imperial_guard_on = villain == imperial_guard;
    assign rocks_on =          villain == rocks;
    assign lava_on =           villain == lava;
    assign snake_on =          villain == snake;

    wire guard_start_timer, guard_timer_expired;
    wire [6:0] guard_timer_value;
    wire [2:0] imperial_guard_state;
    wire [11:0] imperial_guard_pixel;
    //create the imperial guard and its own timer
    Timer imperial_guard_timer(.clk(clock_25mhz),
.start_timer(guard_start_timer), .value(guard_timer_value),
.expired(guard_timer_expired));
    Imperial_Guard imperialguard(.clk(clock_25mhz), .rst(reset),
.pixel_on(imperial_guard_on), .timer_expired(guard_timer_expired),
                                .start_timer(guard_start_timer),
.timer_value(guard_timer_value), .state(imperial_guard_state),
                                .hcount(hcount), .vcount(vcount),
.pixel(imperial_guard_pixel));

    wire snake_start_timer, snake_timer_expired;
    wire [6:0] snake_timer_value;
    wire [2:0] snake_state;
    wire [11:0] snake_pixel;
    //create the imperial guard and its own timer
    Timer snake_timer(.clk(clock_25mhz), .start_timer(snake_start_timer),
.value(snake_timer_value), .expired(snake_timer_expired));
    snake snake_villain(.clk(clock_25mhz), .rst(reset), .pixel_on(snake_on),
.timer_expired(snake_timer_expired),
                                .start_timer(snake_start_timer),
.timer_value(snake_timer_value), .state(snake_state),
                                .hcount(hcount), .vcount(vcount),
.pixel(snake_pixel));

    wire rocks_start_timer, rocks_timer_expired;
    wire [6:0] rocks_timer_value;
    wire [2:0] rocks_state;
    wire [11:0] rocks_pixel;
    //create the imperial guard and its own timer
    Timer rocks_timer(.clk(clock_25mhz), .start_timer(rocks_start_timer),
.value(rocks_timer_value), .expired(rocks_timer_expired));
    Rocks rocks_villain(.clk(clock_25mhz), .pixel_on(rocks_on),
.timer_expired(rocks_timer_expired),
                                    .start_timer(rocks_start_timer),
.timer_value(rocks_timer_value), .state(rocks_state),
                                    .hcount(hcount), .vcount(vcount),
.pixel(rocks_pixel),
                                    .game_level(game_level),
.saved(monkey_saved), .level_state(level_state)); // special inputs



////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////
    //collision detection in this module
    wire bad_collide, good_collide;
    Bad_Collision bad_collisions(.clk(clock_25mhz), .rst(reset),
.player_position(player_position),
                                .villain_state(imperial_guard_state &
{3{imperial_guard_on}} |
                                              rocks_state & {3{rocks_on}} |
                                              snake_state & {3{snake_on}}),
                                .collided(bad_collide));

    Good_Collision good_collisions(.clk(clock_25mhz), .rst(reset),
.pick_or_carpet(pick_carpet), .player_position(player_position),
                                .item_state(apples_state & {3{apples_on}} |
                                            monkey_state & {3{carpet_on}} |
                                            sword_state & {3{sword_on}}),
                                .collided(good_collide));

    wire flash_start_timer, flash_timer_expired;
    wire [6:0] flash_timer_value, flash_counter;
    wire [11:0] flashing_player_pixel;
    flashing flash(.pixel(player_pixel), .clk(clock_25mhz),
.flash(bad_collide), .timer_expired(flash_timer_expired),
                                     .start_timer(flash_start_timer),
.timer_value(flash_timer_value),
                                    .flashing_pixel(flashing_player_pixel));
    Flash_Timer flashTimer(.clk(clock_25mhz), .start_timer(flash_start_timer),
.value(flash_timer_value), .expired(flash_timer_expired),
                           .counter(flash_counter));


////////////////////////////////////////////////////////////////////////////////
////
    //show states on the 7 seg display and LEDS
    assign data[3:0] = game_level;
//    assign data[31:24] = leftshift;
//    assign data[31:28] = game_timer_value;
//    assign data[27:24] = game_counter;
//    assign data[23:20] = latch_sent;
//    assign data[19:16] = pulse_received;
//    assign data[15:12] = data_finalized;
//    assign data[11:8] = A | B | select | nstart | up | down | nleft |
nright;
//    assign data[11:8] = rocks_state;
    assign data[7:4] = level_state;

//    assign LED[0] = imperial_guard_on;
//    assign LED[1] = apples_on;
//    assign LED[2] = rocks_on;
//    assign LED[3] = carpet_on;
//    assign LED[4] = snake_on;
//    assign LED[5] = sword_on;

    assign LED[15] = ongoing;

    assign {LED16_B, LED16_G, LED16_R} = good_collide? 3'b010 :(bad_collide?
3'b001 : 3'b000);



// module responsible for making a scrolling image
// input 'scroll' will control background movement
    wire [9:0] leftshift;
    wire clk_1hz;
    parameter bg_y = 200;

    gen_clk_1hz slow_clk(.clk_25mhz(clock_25mhz), .clk_1hz(clk_1hz));

    move_background mbg (.clock_1hz(clk_1hz),
                         .scroll(player_position),
                         .xloc(leftshift));

// modules that draw the background image
    wire [11:0] background1, background2, bottom_background;
    gen_background bg (.pixel_clk(clock_25mhz),
                       .x(10'd0),
                       .y(bg_y),
                       .hcount(hcount),
                       .vcount(vcount),
                       .leftshift(leftshift),
                       .pixel(background1));

    gen_background bg2 (.pixel_clk(clock_25mhz),
                        .x(10'd320),
                        .y(bg_y),
                        .hcount(hcount),
                        .vcount(vcount),
                        .leftshift(leftshift),
                        .pixel(background2));

    bottom_back bb (.pixel_clk(clock_25mhz),
                    .x(10'd0),
                    .y(bg_y+10'd197),
                    .hcount(hcount),
                    .vcount(vcount),
                    .pixel(bottom_background));

    wire [11:0] background_pixel;
    assign background_pixel = background1 | background2 | bottom_background;


//////////////////////////////////////////////////////////////////////////////
///////////////////
    //instantiate score module and graphics and sound
    wire [15:0] score;
    Score_Unit su(.clk(clock_25mhz), .rst(reset), .ongoing(ongoing),
.good_collision(good_collide), .game_level(game_level),
                .bad_collision(bad_collide), .score(score));
    wire [11:0] score_pixel;
    score_graphics sg(.clk(clock_25mhz), .score(score | SW[10:0]),
.vcount(vcount), .hcount(hcount), .pixel(score_pixel),
                        .level_state(level_state), .game_level(game_level));

    //cover page for level
    wire [11:0] cover_pixel;
    level_cover lc(.clk(clock_25mhz), .level_state(level_state),
.level(game_level), .pixel(cover_pixel),
                    .hcount(hcount), .vcount(vcount));

    //show blobs on screen
    wire [11:0] item_pixel, villain_pixel, output_pixel, dividers_pixel;
    assign item_pixel = apples_pixel | sword_pixel | carpet_pixel |
monkey_pixel;
    assign villain_pixel = imperial_guard_pixel | snake_pixel | rocks_pixel;

    assign output_pixel = villain_pixel | flashing_player_pixel | item_pixel;
    assign dividers_pixel = score_pixel | cover_pixel;//(score_pixel !=
12'd0)? score_pixel: cover_pixel;

    assign {VGA_R, VGA_G, VGA_B} = at_display_area? (ongoing? (((output_pixel
| dividers_pixel) ==0)? background_pixel: output_pixel | dividers_pixel):
dividers_pixel): 0;

    wire sound; //, s2;
    sound_effects SFX( .clk(clock_25mhz), .good_collide(good_collide),
.bad_collide(bad_collide), .frequency(sound));

    sound_gen so (.clk(clock_25mhz), .sound(s2));

    assign JA[0] = sound | (ongoing & s2);


endmodule
```

(kennethc modules)

```
module color_map(
    input [3:0] image_bits,
    output reg [3:0] red, green, blue
    );
    always @ (*)
    begin
        case (image_bits)
            4'd0:  begin red <= 4'b0010; green <= 4'b0001; blue <= 4'b0001;
end
            4'd1:  begin red <= 4'b0100; green <= 4'b0001; blue <= 4'b0000;
end
            4'd2:  begin red <= 4'b1101; green <= 4'b0000; blue <= 4'b0001;
end
            4'd3:  begin red <= 4'b0101; green <= 4'b0010; blue <= 4'b1010;
end
            4'd4:  begin red <= 4'b0101; green <= 4'b0011; blue <= 4'b0111;
end
            4'd5:  begin red <= 4'b0111; green <= 4'b0100; blue <= 4'b0001;
end
            4'd6:  begin red <= 4'b1000; green <= 4'b0011; blue <= 4'b0001;
end
            4'd7:  begin red <= 4'b1010; green <= 4'b0011; blue <= 4'b0000;
end
            4'd8:  begin red <= 4'b0110; green <= 4'b0101; blue <= 4'b0101;
end
            4'd9:  begin red <= 4'b1100; green <= 4'b0100; blue <= 4'b0000;
end
            4'd10: begin red <= 4'b1101; green <= 4'b0111; blue <= 4'b0010;
end
            4'd11: begin red <= 4'b1010; green <= 4'b1000; blue <= 4'b1001;
end
            4'd12: begin red <= 4'b1101; green <= 4'b1001; blue <= 4'b0110;
end
```

```
            4'd13: begin red <= 4'b1110; green <= 4'b1010; blue <= 4'b0010;
end
            4'd14: begin red <= 4'b1101; green <= 4'b1100; blue <= 4'b1011;
end
            4'd15: begin red <= 4'b1011; green <= 4'b1111; blue <= 4'b0010;
end
        endcase
    end
endmodule


module abu_mux(
    input [1:0] im_sel, // choose the image of aladdin to draw
    input [9:0] x, hcount, // images need to know where they are being drawn
    input [9:0] y, vcount,
    input clk, // clock for the generating modules to run off of
    output reg [11:0] pixel // output RGB for appropriate pixel
    );

    wire [11:0] a_sit, a_run1, a_run2;

    gen_abu_sit a_s (.pixel_clk(clk),
                     .x(x),
                     .y(y),
                     .hcount(hcount),
                     .vcount(vcount),
                     .pixel(a_sit));

    gen_abu_run1 a_r1 (.pixel_clk(clk),
                       .x(x),
                       .y(y),
                       .hcount(hcount),
                       .vcount(vcount),
                       .pixel(a_run1));

    gen_abu_run2 a_r2 (.pixel_clk(clk),
                       .x(x),
                       .y(y),
                       .hcount(hcount),
                       .vcount(vcount),
                       .pixel(a_run2));

    always @ (im_sel)
    begin
        case(im_sel)
            2'b00: pixel <= a_sit; // carpet flying
            2'b01: pixel <= a_run1;  // carpet walking (1)
            2'b10: pixel <= a_run2;  // carpet walking (2)
        endcase
    end

endmodule


//////////////////////////////////////////////////////////////////////////////////
////
//
//
//       The modules for each of the image generators is below this point
//
//    Each is the same but pulls from different BRAM. All use same color map
//
//
//
//////////////////////////////////////////////////////////////////////////////////
////

module gen_abu_sit
    #(parameter WIDTH = 36,
                HEIGHT = 22)
    (input pixel_clk,
    input [9:0] x, hcount,
    input [9:0] y, vcount, // x and y denote the (x,y) of the top left corner
of the image
    output reg [11:0] pixel
    );

    wire [9:0] image_addr; // retrieving color information for a pixel
    wire [3:0] image_bits; // 4 bits that encode a particular pixel
    wire [3:0] r, g, b; // 12 bits total for VGA
    reg [9:0] phcount, pvcount; // pipelined hcount and vcount

    always @ (posedge pixel_clk)
    begin
        if ((phcount >= x && phcount < (x+WIDTH)) && (pvcount >= y && pvcount
< (y+HEIGHT)))
            case (g)
                4'b1111: pixel <= 0;
                default: pixel <= {r,g,b};
            endcase
        // else pixel = DEFAULT;
        else pixel = 0;
        phcount <= hcount;
        pvcount <= vcount;
    end
    assign image_addr = (hcount - x) + (vcount - y) * WIDTH;
    abu_sit a_sit(.addra(image_addr), .clka(pixel_clk), .douta(image_bits));

    color_map cm(image_bits, r, g, b); // outputs appropriate RGB values for
pixel
endmodule

module gen_abu_run1
    #(parameter WIDTH = 36,
                HEIGHT = 22)
    (input pixel_clk,
    input [9:0] x, hcount,
    input [9:0] y, vcount, // x and y denote the (x,y) of the top left corner
of the image
    output reg [11:0] pixel
    );

    wire [9:0] image_addr; // retrieving color information for a pixel
    wire [3:0] image_bits; // 4 bits that encode a particular pixel
    wire [3:0] r, g, b; // 12 bits total for VGA
    reg [9:0] phcount, pvcount; // pipelined hcount and vcount

    always @ (posedge pixel_clk)
    begin
        if ((phcount >= x && phcount < (x+WIDTH)) && (pvcount >= y && pvcount
< (y+HEIGHT)))
            case (g)
                4'b1111: pixel <= 0;
                default: pixel <= {r,g,b};
            endcase
        // else pixel = DEFAULT;
        else pixel = 0;
        phcount <= hcount;
        pvcount <= vcount;
    end

    assign image_addr = (hcount - x) + (vcount - y) * WIDTH;
    abu_run1 a_run(.addra(image_addr), .clka(pixel_clk), .douta(image_bits));

    color_map cm(image_bits, r, g, b); // outputs appropriate RGB values for
pixel
endmodule

module gen_abu_run2
    #(parameter WIDTH = 36,
                HEIGHT = 22)
    (input pixel_clk,
    input [9:0] x, hcount,
    input [9:0] y, vcount, // x and y denote the (x,y) of the top left corner
of the image
    output reg [11:0] pixel
    );

    wire [9:0] image_addr; // retrieving color information for a pixel
    wire [3:0] image_bits; // 4 bits that encode a particular pixel
    wire [3:0] r, g, b; // 12 bits total for VGA
    reg [9:0] phcount, pvcount; // pipelined hcount and vcount

    always @ (posedge pixel_clk)
    begin
        if ((phcount >= x && phcount < (x+WIDTH)) && (pvcount >= y && pvcount
< (y+HEIGHT)))
            case (g)
                4'b1111: pixel <= 0;
                default: pixel <= {r,g,b};
            endcase
        // else pixel = DEFAULT;
        else pixel = 0;
        phcount <= hcount;
        pvcount <= vcount;
    end

    assign image_addr = (hcount - x) + (vcount - y) * WIDTH;
    abu_run2 a_run2(.addra(image_addr), .clka(pixel_clk), .douta(image_bits));

    color_map cm(image_bits, r, g, b); // outputs appropriate RGB values for
pixel
endmodule



// NOTE: Each picture of Aladdin is 37x57
// Create separate BRAMS and modules for each aladdin
parameter DEFAULT = 0;

module aladdin_mux
    (input [2:0] im_sel, // choose the image of aladdin to draw
     input [9:0] x, hcount, // images need to know where they are being drawn
     input [9:0] y, vcount,
     input clk, // clock for the generating modules to run off of
     output reg [11:0] pixel // output RGB for appropriate pixel
    );

    wire [11:0] p_st, p_r1, p_r2, p_r3, p_r4, p_gd, p_gu, p_t1, p_t2; // have
12 bit wires for each possible image
```

```
    gen_aladdin_st a_st(.pixel_clk(clk),
                         .x(x),
                         .y(y),
                         .hcount(hcount),
                         .vcount(vcount),
                         .pixel(p_st));

    gen_aladdin_r1 a_r1(.pixel_clk(clk),
                        .x(x),
                        .y(y),
                        .hcount(hcount),
                        .vcount(vcount),
                        .pixel(p_r1));

    gen_aladdin_r2 a_r2(.pixel_clk(clk),
                        .x(x),
                        .y(y),
                        .hcount(hcount),
                        .vcount(vcount),
                        .pixel(p_r2));

//    gen_aladdin_r3 a_r3(.pixel_clk(clk),
//                        .x(x),
//                        .y(y),
//                        .hcount(hcount),
//                        .vcount(vcount),
//                        .pixel(p_r3));

//    gen_aladdin_r4 a_r4(.pixel_clk(clk),
//                        .x(x),
//                        .y(y),
//                        .hcount(hcount),
//                        .vcount(vcount),
//                        .pixel(p_r4));

    gen_aladdin_gd a_gd(.pixel_clk(clk),
                        .x(x),
                        .y(y),
                        .hcount(hcount),
                        .vcount(vcount),
                        .pixel(p_gd));

    gen_aladdin_gu a_gu(.pixel_clk(clk),
                        .x(x),
                        .y(y),
                        .hcount(hcount),
                        .vcount(vcount),
                        .pixel(p_gu));
    gen_aladdin_t1 a_t1(.pixel_clk(clk),
                        .x(x),
                        .y(y),
                        .hcount(hcount),
                        .vcount(vcount),
                        .pixel(p_t1));

    gen_aladdin_t2 a_t2(.pixel_clk(clk),
                        .x(x),
                        .y(y),
                        .hcount(hcount),
                        .vcount(vcount),
                        .pixel(p_t2));

    always @ (im_sel) // for generating a pictures
    begin
        case(im_sel)
            3'b000: pixel <= p_st; // for when he is standing

            3'b001: pixel <= p_r1; // sprites for running (ONLY NEED THESE
TWO, BUT YOU CAN USE MORE IF YOU WANT)
            3'b010: pixel <= p_r2;

            3'b011: pixel <= p_gd; // sprites for grabbing
            3'b100: pixel <= p_gu;

            3'b101: pixel <= p_t1; // sprites for throwing
            3'b110: pixel <= p_t2;

        endcase
    end
endmodule



//////////////////////////////////////////////////////////////////////////////
////
//
//
//       The modules for each of the image generators is below this point
//
//    Each is the same but pulls from different BRAM. All use same color map
//
//
//
//////////////////////////////////////////////////////////////////////////////
////
```

```
// standing figure for aladdin
module gen_aladdin_st
    #(parameter WIDTH = 37,
                HEIGHT = 57)
    (input pixel_clk,
    input [9:0] x, hcount,
    input [9:0] y, vcount, // x and y denote the (x,y) of the top left corner
of the image
    output reg [11:0] pixel
    );

    wire [11:0] image_addr; // retrieving color information for a pixel
    wire [3:0] image_bits; // 4 bits that encode a particular pixel
    wire [3:0] r, g, b; // 12 bits total for VGA
    reg [9:0] phcount, pvcount; // pipelined hcount and vcount

    always @ (posedge pixel_clk)
    begin
        if ((phcount >= x && phcount < (x+WIDTH)) && (pvcount >= y && pvcount
< (y+HEIGHT)))
            case (g)
                4'b1111: pixel <= 0;
                default: pixel <= {r,g,b};
            endcase
        // else pixel = DEFAULT;
        else pixel = 0;
        phcount <= hcount;
        pvcount <= vcount;
    end

    assign image_addr = (hcount - x) + (vcount - y) * WIDTH;
    aladdin_stand aladdin_st(.addra(image_addr), .clka(pixel_clk),
.douta(image_bits));

    color_map cm(image_bits, r, g, b); // outputs appropriate RGB values for
pixel
endmodule


// first running figure for aladdin
module gen_aladdin_r1
    #(parameter WIDTH = 37,
                HEIGHT = 57)
    (input pixel_clk,
    input [9:0] x, hcount,
    input [9:0] y, vcount, // x and y denote the (x,y) of the top left corner
of the image
```

```
    output reg [11:0] pixel
    );

    wire [11:0] image_addr; // retrieving color information for a pixel
    wire [3:0] image_bits; // 4 bits that encode a particular pixel
    wire [3:0] r, g, b; // 12 bits total for VGA
    reg [9:0] phcount, pvcount; // pipelined hcount and vcount

    always @ (posedge pixel_clk)
    begin
        if ((phcount >= x && phcount < (x+WIDTH)) && (pvcount >= y && pvcount
< (y+HEIGHT)))
            case (g)
                4'b1111: pixel <= 0;
                default: pixel <= {r,g,b};
            endcase
        // else pixel = DEFAULT;
        else pixel = 0;
        phcount <= hcount;
        pvcount <= vcount;
    end

    assign image_addr = (hcount - x) + (vcount - y) * WIDTH;
    aladdin_run1 aladdin_r1(.addra(image_addr), .clka(pixel_clk),
.douta(image_bits));

    color_map cm(image_bits, r, g, b); // find the corresponding rgb values
endmodule


// second running figure for aladdin
module gen_aladdin_r2
    #(parameter WIDTH = 37,
                HEIGHT = 57)
    (input pixel_clk,
    input [9:0] x, hcount,
    input [9:0] y, vcount, // x and y denote the (x,y) of the top left corner
of the image
    output reg [11:0] pixel
    );

    wire [11:0] image_addr; // retrieving color information for a pixel
    wire [3:0] image_bits; // 4 bits that encode a particular pixel
    wire [3:0] r, g, b; // 12 bits total for VGA
    reg [9:0] phcount, pvcount; // pipelined hcount and vcount

    always @ (posedge pixel_clk)
    begin
```

```
        if ((phcount >= x && phcount < (x+WIDTH)) && (pvcount >= y && pvcount
< (y+HEIGHT)))
            case (g)
                4'b1111: pixel <= 0;
                default: pixel <= {r,g,b};
            endcase
        // else pixel = DEFAULT;
        else pixel = 0;
        phcount <= hcount;
        pvcount <= vcount;
    end

    assign image_addr = (hcount - x) + (vcount - y) * WIDTH;
    aladdin_run2 aladdin_r2(.addra(image_addr), .clka(pixel_clk),
.douta(image_bits));

    color_map cm(image_bits, r, g, b); // find the corresponding rgb values
endmodule

//module gen_aladdin_r3
//    #(parameter WIDTH = 37,
//                HEIGHT = 57)
//    (input pixel_clk,
//    input [9:0] x, hcount,
//    input [9:0] y, vcount, // x and y denote the (x,y) of the top left
corner of the image
//    output reg [11:0] pixel
//    );

//    wire [11:0] image_addr; // retrieving color information for a pixel
//    wire [3:0] image_bits; // 4 bits that encode a particular pixel
//    wire [3:0] r, g, b; // 12 bits total for VGA
//    reg [9:0] phcount, pvcount; // pipelined hcount and vcount

//    always @ (posedge pixel_clk)
//    begin
//        if ((phcount >= x && phcount < (x+WIDTH)) && (pvcount >= y &&
pvcount < (y+HEIGHT)))
//            pixel <= {r,g,b};
//        else pixel = DEFAULT;
//        phcount <= hcount;
//        pvcount <= vcount;
//    end

//    assign image_addr = (hcount - x) + (vcount - y) * WIDTH;
//    aladdin_run3 aladdin_r3(.addra(image_addr), .clka(pixel_clk),
.douta(image_bits));
```

```
//    color_map cm(image_bits, r, g, b); // find the corresponding rgb values
//endmodule

//module gen_aladdin_r4
//    #(parameter WIDTH = 37,
//                HEIGHT = 57)
//    (input pixel_clk,
//    input [9:0] x, hcount,
//    input [9:0] y, vcount, // x and y denote the (x,y) of the top left
corner of the image
//    output reg [11:0] pixel
//    );

//    wire [11:0] image_addr; // retrieving color information for a pixel
//    wire [3:0] image_bits; // 4 bits that encode a particular pixel
//    wire [3:0] r, g, b; // 12 bits total for VGA
//    reg [9:0] phcount, pvcount; // pipelined hcount and vcount

//    always @ (posedge pixel_clk)
//    begin
//        if ((phcount >= x && phcount < (x+WIDTH)) && (pvcount >= y &&
pvcount < (y+HEIGHT)))
//            pixel <= {r,g,b};
//        else pixel = DEFAULT;
//        phcount <= hcount;
//        pvcount <= vcount;
//    end

//    assign image_addr = (hcount - x) + (vcount - y) * WIDTH;
//    aladdin_run4 aladdin_r4(.addra(image_addr), .clka(pixel_clk),
.douta(image_bits));

//    color_map cm(image_bits, r, g, b); // find the corresponding rgb values
//endmodule

module gen_aladdin_gd
    #(parameter WIDTH = 37,
                HEIGHT = 57)
    (input pixel_clk,
    input [9:0] x, hcount,
    input [9:0] y, vcount, // x and y denote the (x,y) of the top left corner
of the image
    output reg [11:0] pixel
    );

    wire [11:0] image_addr; // retrieving color information for a pixel
    wire [3:0] image_bits; // 4 bits that encode a particular pixel
    wire [3:0] r, g, b; // 12 bits total for VGA
    reg [9:0] phcount, pvcount; // pipelined hcount and vcount

    always @ (posedge pixel_clk)
    begin
        if ((phcount >= x && phcount < (x+WIDTH)) && (pvcount >= y && pvcount
< (y+HEIGHT)))
            case (g)
                4'b1111: pixel <= 0;
                default: pixel <= {r,g,b};
            endcase
        // else pixel = DEFAULT;
        else pixel = 0;
        phcount <= hcount;
        pvcount <= vcount;
    end

    assign image_addr = (hcount - x) + (vcount - y) * WIDTH;
    aladdin_grab_down aladdin_gd(.addra(image_addr), .clka(pixel_clk),
.douta(image_bits));

    color_map cm(image_bits, r, g, b); // find the corresponding rgb values
endmodule

module gen_aladdin_gu
    #(parameter WIDTH = 37,
                HEIGHT = 57)
    (input pixel_clk,
    input [9:0] x, hcount,
    input [9:0] y, vcount, // x and y denote the (x,y) of the top left corner
of the image
    output reg [11:0] pixel
    );

    wire [11:0] image_addr; // retrieving color information for a pixel
    wire [3:0] image_bits; // 4 bits that encode a particular pixel
    wire [3:0] r, g, b; // 12 bits total for VGA
    reg [9:0] phcount, pvcount; // pipelined hcount and vcount

    always @ (posedge pixel_clk)
    begin
        if ((phcount >= x && phcount < (x+WIDTH)) && (pvcount >= y && pvcount
< (y+HEIGHT)))
            case (g)
                4'b1111: pixel <= 0;
                default: pixel <= {r,g,b};
            endcase
        // else pixel = DEFAULT;
        else pixel = 0;
        phcount <= hcount;
        pvcount <= vcount;
    end

    assign image_addr = (hcount - x) + (vcount - y) * WIDTH;
    aladdin_grab_up aladdin_gu(.addra(image_addr), .clka(pixel_clk),
.douta(image_bits));

    color_map cm(image_bits, r, g, b); // find the corresponding rgb values
endmodule

module gen_aladdin_t1
    #(parameter WIDTH = 37,
                HEIGHT = 57)
    (input pixel_clk,
    input [9:0] x, hcount,
    input [9:0] y, vcount, // x and y denote the (x,y) of the top left corner
of the image
    output reg [11:0] pixel
    );

    wire [11:0] image_addr; // retrieving color information for a pixel
    wire [3:0] image_bits; // 4 bits that encode a particular pixel
    wire [3:0] r, g, b; // 12 bits total for VGA
    reg [9:0] phcount, pvcount; // pipelined hcount and vcount

    always @ (posedge pixel_clk)
    begin
        if ((phcount >= x && phcount < (x+WIDTH)) && (pvcount >= y && pvcount
< (y+HEIGHT)))
            case (g)
                4'b1111: pixel <= 0;
                default: pixel <= {r,g,b};
            endcase
        // else pixel = DEFAULT;
        else pixel = 0;
        phcount <= hcount;
        pvcount <= vcount;
    end

    assign image_addr = (hcount - x) + (vcount - y) * WIDTH;
    aladdin_throw_1 aladdin_t1(.addra(image_addr), .clka(pixel_clk),
.douta(image_bits));

    color_map cm(image_bits, r, g, b); // find the corresponding rgb values
endmodule

module gen_aladdin_t2
```

```
    #(parameter WIDTH = 37,
                HEIGHT = 57)
    (input pixel_clk,
    input [9:0] x, hcount,
    input [9:0] y, vcount, // x and y denote the (x,y) of the top left corner
of the image
    output reg [11:0] pixel
    );

    wire [11:0] image_addr; // retrieving color information for a pixel
    wire [3:0] image_bits; // 4 bits that encode a particular pixel
    wire [3:0] r, g, b; // 12 bits total for VGA
    reg [9:0] phcount, pvcount; // pipelined hcount and vcount

    always @ (posedge pixel_clk)
    begin
        if ((phcount >= x && phcount < (x+WIDTH)) && (pvcount >= y && pvcount
< (y+HEIGHT)))
            case (g)
                4'b1111: pixel <= 0;
                default: pixel <= {r,g,b};
            endcase
        // else pixel = DEFAULT;
        else pixel = 0;
        phcount <= hcount;
        pvcount <= vcount;
    end

    assign image_addr = (hcount - x) + (vcount - y) * WIDTH;
    aladdin_throw_2 aladdin_t2(.addra(image_addr), .clka(pixel_clk),
.douta(image_bits));

    color_map cm(image_bits, r, g, b); // find the corresponding rgb values
endmodule

module gen_background
    #(parameter WIDTH = 320,
                HEIGHT = 200)
    (input pixel_clk,
    input [9:0] x, hcount,
    input [9:0] y, vcount, // x and y denote the (x,y) of the top left corner
of the image
    input [9:0] leftshift,
    output reg [11:0] pixel
    );

    wire [16:0] image_addr; // retrieving color information for a pixel
    wire [3:0] image_bits; // 4 bits that encode a particular pixel
    wire [3:0] r, g, b; // 12 bits total for VGA
    reg [9:0] phcount, pvcount; // pipelined hcount and vcount
    reg [9:0] offset;

    always @ (posedge pixel_clk)
    begin
        if ((phcount >= x && phcount < (x+WIDTH)) && (pvcount >= y && pvcount
< (y+HEIGHT)))
            pixel <= {r,g,b};
        else pixel = 0;
        phcount <= hcount;
        pvcount <= vcount;
    end

    assign image_addr = (hcount - x + leftshift) + (vcount - y) * WIDTH;
    background3_new b3(.addra(image_addr), .clka(pixel_clk),
.douta(image_bits));

    color_map cm(image_bits, r, g, b); // find the corresponding rgb values
endmodule

module bottom_back
    #(parameter WIDTH = 640,
                HEIGHT = 20,
                COLOR = 12'b111010100010)
    (input pixel_clk,
    input [9:0] x, hcount,
    input [9:0] y, vcount, // x and y denote the (x,y) of the top left corner
of the image
    output reg [11:0] pixel
    );

    reg [9:0] phcount, pvcount; // pipelined hcount and vcount

    always @ (posedge pixel_clk)
    begin
        if ((phcount >= x && phcount < (x+WIDTH)) && (pvcount >= y && pvcount
< (y+HEIGHT)))
            pixel <= COLOR;
        else pixel = 0;
        phcount <= hcount;
        pvcount <= vcount;
    end
endmodule

module move_background
    (input clock_1hz,
     input scroll,
     output [9:0] xloc);

     reg [9:0] x = 0;

     always @ (posedge clock_1hz)
     begin
        if (x >= 320) x <= 0;
        else if (scroll) x <= x + 20;
     end

     assign xloc = x;

endmodule

//parameter DEFAULT = 12'hFFF;


module badguy_mux
    (input [2:0] im_sel, // choose the image of aladdin to draw
     input [9:0] x, hcount, // images need to know where they are being drawn
     input [9:0] y, vcount,
     input clk, // clock for the generating modules to run off of
     output reg [11:0] pixel // output RGB for appropriate pixel
    );

    wire [11:0] gp_st, gp_at, gp_def, jp, jp_sn; // have 12 bit wires for each
possible image

    gen_guard_st g_st(.pixel_clk(clk),
                      .x(x),
                      .y(y),
                      .hcount(hcount),
                      .vcount(vcount),
                      .pixel(gp_st));

    gen_guard_at g_at(.pixel_clk(clk),
                      .x(x),
                      .y(y),
                      .hcount(hcount),
                      .vcount(vcount),
                      .pixel(gp_at));

    gen_guard_def g_def(.pixel_clk(clk),
                        .x(x),
                        .y(y),
                        .hcount(hcount),
                        .vcount(vcount),
                        .pixel(gp_def));
    gen_jafar jaf(.pixel_clk(clk),
                  .x(x),
                  .y(y),
                  .hcount(hcount),
                  .vcount(vcount),
                  .pixel(jp));

    gen_jafar_snake j_sn(.pixel_clk(clk),
                         .x(x),
                         .y(y),
                         .hcount(hcount),
                         .vcount(vcount),
                         .pixel(jp_sn));

    always @ (im_sel)
    begin
        case(im_sel)
            3'b000: pixel <= gp_st; // guard standing
            3'b001: pixel <= gp_at; // guard attacking
            3'b010: pixel <= gp_def; // guard defeated

            3'b011: pixel <= jp; // jafar
            3'b100: pixel <= jp_sn; // jafar snake

        endcase
    end
endmodule

//////////////////////////////////////////////////////////////////////////////
////
//
//
//       The modules for each of the image generators is below this point
//
//    Each is the same but pulls from different BRAM. All use same color map
//
//
//
//////////////////////////////////////////////////////////////////////////////
////

module gen_guard_st
    #(parameter WIDTH = 67,
                HEIGHT = 65)
    (input pixel_clk,
    input [9:0] x, hcount,
    input [9:0] y, vcount, // x and y denote the (x,y) of the top left corner
of the image
    output reg [11:0] pixel
    );

    wire [12:0] image_addr; // retrieving color information for a pixel
    wire [3:0] image_bits; // 4 bits that encode a particular pixel
    wire [3:0] r, g, b; // 12 bits total for VGA
    reg [9:0] phcount, pvcount; // pipelined hcount and vcount

    always @ (posedge pixel_clk)
    begin
        if ((phcount >= x && phcount < (x+WIDTH)) && (pvcount >= y && pvcount
< (y+HEIGHT)))
            case (g)
                4'b1111: pixel <= 0;
                default: pixel <= {r,g,b};
            endcase
        // else pixel = DEFAULT;
        else pixel = 0;
        phcount <= hcount;
        pvcount <= vcount;
    end

    assign image_addr = (hcount - x) + (vcount - y) * WIDTH;
    imperial_guard guard(.addra(image_addr), .clka(pixel_clk),
.douta(image_bits));

    color_map cm(image_bits, r, g, b); // outputs appropriate RGB values for
pixel
endmodule

module gen_guard_at
    #(parameter WIDTH = 67,
                HEIGHT = 65)
    (input pixel_clk,
    input [9:0] x, hcount,
    input [9:0] y, vcount, // x and y denote the (x,y) of the top left corner
of the image
    output reg [11:0] pixel
    );

    wire [12:0] image_addr; // retrieving color information for a pixel
    wire [3:0] image_bits; // 4 bits that encode a particular pixel
    wire [3:0] r, g, b; // 12 bits total for VGA
    reg [9:0] phcount, pvcount; // pipelined hcount and vcount

    always @ (posedge pixel_clk)
    begin
        if ((phcount >= x && phcount < (x+WIDTH)) && (pvcount >= y && pvcount
< (y+HEIGHT)))
            case (g)
                4'b1111: pixel <= 0;
                default: pixel <= {r,g,b};
            endcase
        // else pixel = DEFAULT;
        else pixel = 0;
        phcount <= hcount;
        pvcount <= vcount;
    end

    assign image_addr = (hcount - x) + (vcount - y) * WIDTH;
    imperial_guard_attack guard_at(.addra(image_addr), .clka(pixel_clk),
.douta(image_bits));

    color_map cm(image_bits, r, g, b); // outputs appropriate RGB values for
pixel
endmodule

module gen_guard_def
    #(parameter WIDTH = 67,
                HEIGHT = 65)
    (input pixel_clk,
    input [9:0] x, hcount,
    input [9:0] y, vcount, // x and y denote the (x,y) of the top left corner
of the image
    output reg [11:0] pixel
    );

    wire [12:0] image_addr; // retrieving color information for a pixel
    wire [3:0] image_bits; // 4 bits that encode a particular pixel
    wire [3:0] r, g, b; // 12 bits total for VGA
    reg [9:0] phcount, pvcount; // pipelined hcount and vcount

    always @ (posedge pixel_clk)
    begin
        if ((phcount >= x && phcount < (x+WIDTH)) && (pvcount >= y && pvcount
< (y+HEIGHT)))
            case (g)
                4'b1111: pixel <= 0;
                default: pixel <= {r,g,b};
            endcase
        // else pixel = DEFAULT;
        else pixel = 0;
        phcount <= hcount;
        pvcount <= vcount;
    end

    assign image_addr = (hcount - x) + (vcount - y) * WIDTH;
    imperial_guard_defeated guard_def(.addra(image_addr), .clka(pixel_clk),
.douta(image_bits));

    color_map cm(image_bits, r, g, b); // outputs appropriate RGB values for
pixel
endmodule

module gen_jafar
    #(parameter WIDTH = 67,
                HEIGHT = 65)
    (input pixel_clk,
    input [9:0] x, hcount,
    input [9:0] y, vcount, // x and y denote the (x,y) of the top left corner
of the image
    output reg [11:0] pixel
    );

    wire [12:0] image_addr; // retrieving color information for a pixel
    wire [3:0] image_bits; // 4 bits that encode a particular pixel
    wire [3:0] r, g, b; // 12 bits total for VGA
    reg [9:0] phcount, pvcount; // pipelined hcount and vcount

    always @ (posedge pixel_clk)
    begin
        if ((phcount >= x && phcount < (x+WIDTH)) && (pvcount >= y && pvcount
< (y+HEIGHT)))
            case (g)
                4'b1111: pixel <= 0;
                default: pixel <= {r,g,b};
            endcase
        // else pixel = DEFAULT;
        else pixel = 0;
        phcount <= hcount;
        pvcount <= vcount;
    end

    assign image_addr = (hcount - x) + (vcount - y) * WIDTH;
    jafar j(.addra(image_addr), .clka(pixel_clk), .douta(image_bits));

    color_map cm(image_bits, r, g, b); // outputs appropriate RGB values for
pixel
endmodule

module gen_jafar_snake
    #(parameter WIDTH = 67,
                HEIGHT = 65)
    (input pixel_clk,
    input [9:0] x, hcount,
    input [9:0] y, vcount, // x and y denote the (x,y) of the top left corner
of the image
    output reg [11:0] pixel
    );

    wire [12:0] image_addr; // retrieving color information for a pixel
    wire [3:0] image_bits; // 4 bits that encode a particular pixel
    wire [3:0] r, g, b; // 12 bits total for VGA
    reg [9:0] phcount, pvcount; // pipelined hcount and vcount

    always @ (posedge pixel_clk)
    begin
        if ((phcount >= x && phcount < (x+WIDTH)) && (pvcount >= y && pvcount
< (y+HEIGHT)))
            case (g)
                4'b1111: pixel <= 0;
                default: pixel <= {r,g,b};
            endcase
        // else pixel = DEFAULT;
        else pixel = 0;
        phcount <= hcount;
        pvcount <= vcount;
    end

    assign image_addr = (hcount - x) + (vcount - y) * WIDTH;
    jafar_snake j_snake(.addra(image_addr), .clka(pixel_clk),
.douta(image_bits));

    color_map cm(image_bits, r, g, b); // outputs appropriate RGB values for
pixel
endmodule


module carpet_mux(
     input [1:0] im_sel, // choose the image of aladdin to draw
     input [9:0] x, hcount, // images need to know where they are being drawn
     input [9:0] y, vcount,
     input clk, // clock for the generating modules to run off of
     output reg [11:0] pixel // output RGB for appropriate pixel
    );

    wire [11:0] c_fly, c_w1, c_w2;

    gen_carpet_fly cf (.pixel_clk(clk),
                       .x(x),
                       .y(y),
                       .hcount(hcount),
                       .vcount(vcount),
                       .pixel(c_fly));

    gen_carpet_walk1 cw1 (.pixel_clk(clk),
                          .x(x),
                          .y(y),
                          .hcount(hcount),
                          .vcount(vcount),
                          .pixel(c_w1));

    gen_carpet_walk2 cw2 (.pixel_clk(clk),
                          .x(x),
                          .y(y),
                          .hcount(hcount),
                          .vcount(vcount),
                          .pixel(c_w2));

    always @ (im_sel)
        begin
            case(im_sel)
                2'b00: pixel <= c_fly; // carpet flying
                2'b01: pixel <= c_w1;  // carpet walking (1)
                2'b11: pixel <= c_w2;  // carpet walking (2)
            endcase
        end
endmodule


//////////////////////////////////////////////////////////////////////////////
////
//
//
//       The modules for each of the image generators is below this point
//
//    Each is the same but pulls from different BRAM. All use same color map
//
//
//
//////////////////////////////////////////////////////////////////////////////
////

module gen_carpet_fly
    #(parameter WIDTH = 33,
                HEIGHT = 32)
    (input pixel_clk,
    input [9:0] x, hcount,
    input [9:0] y, vcount, // x and y denote the (x,y) of the top left corner
of the image
    output reg [11:0] pixel
    );

    wire [10:0] image_addr; // retrieving color information for a pixel
    wire [3:0] image_bits; // 4 bits that encode a particular pixel
    wire [3:0] r, g, b; // 12 bits total for VGA
    reg [9:0] phcount, pvcount; // pipelined hcount and vcount

    always @ (posedge pixel_clk)
    begin
        if ((phcount >= x && phcount < (x+WIDTH)) && (pvcount >= y && pvcount
< (y+HEIGHT)))
            case (g)
                4'b1111: pixel <= 0;
                default: pixel <= {r,g,b};
            endcase
        // else pixel = DEFAULT;
        else pixel = 0;
        phcount <= hcount;
        pvcount <= vcount;
    end

    assign image_addr = (hcount - x) + (vcount - y) * WIDTH;
    carpet_fly c_fly(.addra(image_addr), .clka(pixel_clk),
.douta(image_bits));

    color_map cm(image_bits, r, g, b); // outputs appropriate RGB values for
pixel
endmodule


module gen_carpet_walk1
    #(parameter WIDTH = 33,
                HEIGHT = 32)
    (input pixel_clk,
    input [9:0] x, hcount,
    input [9:0] y, vcount, // x and y denote the (x,y) of the top left corner
of the image
    output reg [11:0] pixel
    );

    wire [10:0] image_addr; // retrieving color information for a pixel
    wire [3:0] image_bits; // 4 bits that encode a particular pixel
    wire [3:0] r, g, b; // 12 bits total for VGA
    reg [9:0] phcount, pvcount; // pipelined hcount and vcount
    always @ (posedge pixel_clk)
    begin
        if ((phcount >= x && phcount < (x+WIDTH)) && (pvcount >= y && pvcount
< (y+HEIGHT)))
            case (g)
                4'b1111: pixel <= 0;
                default: pixel <= {r,g,b};
            endcase
        // else pixel = DEFAULT;
        else pixel = 0;
        phcount <= hcount;
        pvcount <= vcount;
    end

    assign image_addr = (hcount - x) + (vcount - y) * WIDTH;
    capet_walk1 c_w1(.addra(image_addr), .clka(pixel_clk),
.douta(image_bits));

    color_map cm(image_bits, r, g, b); // outputs appropriate RGB values for
pixel
endmodule

module gen_carpet_walk2
    #(parameter WIDTH = 33,
                HEIGHT = 32)
    (input pixel_clk,
    input [9:0] x, hcount,
    input [9:0] y, vcount, // x and y denote the (x,y) of the top left corner
of the image
    output reg [11:0] pixel
    );

    wire [10:0] image_addr; // retrieving color information for a pixel
    wire [3:0] image_bits; // 4 bits that encode a particular pixel
    wire [3:0] r, g, b; // 12 bits total for VGA
    reg [9:0] phcount, pvcount; // pipelined hcount and vcount

    always @ (posedge pixel_clk)
    begin
        if ((phcount >= x && phcount < (x+WIDTH)) && (pvcount >= y && pvcount
< (y+HEIGHT)))
            case (g)
                4'b1111: pixel <= 0;
                default: pixel <= {r,g,b};
            endcase
        // else pixel = DEFAULT;
        else pixel = 0;
        phcount <= hcount;
        pvcount <= vcount;
    end

    assign image_addr = (hcount - x) + (vcount - y) * WIDTH;
    carpet_walk2 c_w2(.addra(image_addr), .clka(pixel_clk),
.douta(image_bits));

    color_map cm(image_bits, r, g, b); // outputs appropriate RGB values for
pixel
endmodule

module kcolor_map
    (input [3:0] image_bits,
     output reg [3:0] red, green, blue);

     always @ (*)
     begin
        case (image_bits)
            4'd0:  begin red <= 4'b0000; green <= 4'b0000; ablue <= 4'b0000;
end
            4'd1:  begin red <= 4'b0011; green <= 4'b0010; blue <= 4'b0010;
end
            4'd2:  begin red <= 4'b0011; green <= 4'b0100; blue <= 4'b0011;
end
            4'd3:  begin red <= 4'b1000; green <= 4'b0110; blue <= 4'b0011;
end
            4'd4:  begin red <= 4'b0111; green <= 4'b0111; blue <= 4'b0110;
end
            4'd5:  begin red <= 4'b0111; green <= 4'b0111; blue <= 4'b0111;
end
            4'd6:  begin red <= 4'b1011; green <= 4'b0111; blue <= 4'b0100;
end
            4'd7:  begin red <= 4'b1011; green <= 4'b0111; blue <= 4'b1000;
end
            4'd8:  begin red <= 4'b1001; green <= 4'b1001; blue <= 4'b1001;
end
            4'd9:  begin red <= 4'b0101; green <= 4'b1011; blue <= 4'b0100;
end
            4'd10: begin red <= 4'b1010; green <= 4'b1001; blue <= 4'b1001;
end
            4'd11: begin red <= 4'b1110; green <= 4'b1010; blue <= 4'b0111;
end
            4'd12: begin red <= 4'b1111; green <= 4'b1010; blue <= 4'b1001;
end
            4'd13: begin red <= 4'b1000; green <= 4'b1111; blue <= 4'b0101;
end
            4'd14: begin red <= 4'b1111; green <= 4'b1101; blue <= 4'b1101;
end
```

```
            4'd15: begin red <= 4'b1111; green <= 4'b1111; blue <= 4'b1110;
end
        endcase
     end
endmodule



module gen_koala
    #(parameter WIDTH = 325,
                HEIGHT = 403)
    (input pixel_clk,
    input [9:0] x, hcount,
    input [9:0] y, vcount, // x and y denote the (x,y) of the top left corner
of the image
    output reg [11:0] pixel
    );

    wire [16:0] image_addr; // retrieving color information for a pixel
    wire [3:0] image_bits; // 4 bits that encode a particular pixel
    wire [3:0] r, g, b; // 12 bits total for VGA
    reg [9:0] phcount, pvcount; // pipelined hcount and vcount

    always @ (posedge pixel_clk)
    begin
        if ((phcount >= x && phcount < (x+WIDTH)) && (pvcount >= y && pvcount
< (y+HEIGHT)))
            pixel <= {r,g,b};
        else pixel = 0;
        phcount <= hcount;
        pvcount <= vcount;
    end

    assign image_addr = (hcount - x) + (vcount - y) * WIDTH;
    koala k(.addra(image_addr), .clka(pixel_clk), .douta(image_bits));

    kcolor_map cm(image_bits, r, g, b); // outputs appropriate RGB values for
pixel
endmodule


module gen_gameover
    #(parameter WIDTH = 600,
                HEIGHT = 60)
    (input pixel_clk,
    input [9:0] x, hcount,
    input [9:0] y, vcount, // x and y denote the (x,y) of the top left corner
of the image
```

```
    output reg [11:0] pixel
    );

    wire [16:0] image_addr; // retrieving color information for a pixel
    wire [3:0] image_bits; // 4 bits that encode a particular pixel
    reg [9:0] phcount, pvcount; // pipelined hcount and vcount

    always @ (posedge pixel_clk)
    begin
        if ((phcount >= x && phcount < (x+WIDTH)) && (pvcount >= y && pvcount
< (y+HEIGHT)))
        begin
            if (image_bits != 0) pixel <= 12'hF00;
            else pixel <= 0;
        end
        else pixel = 0;
        phcount <= hcount;
        pvcount <= vcount;
    end

    assign image_addr = (hcount - x) + (vcount - y) * WIDTH;
    gameover go(.addra(image_addr), .clka(pixel_clk), .douta(image_bits));

endmodule


module gen_start
    #(parameter WIDTH = 600,
                HEIGHT = 60)
    (input pixel_clk,
    input [9:0] x, hcount,
    input [9:0] y, vcount, // x and y denote the (x,y) of the top left corner
of the image
    output reg [11:0] pixel
    );

    wire [16:0] image_addr; // retrieving color information for a pixel
    wire [3:0] image_bits; // 4 bits that encode a particular pixel
    reg [9:0] phcount, pvcount; // pipelined hcount and vcount

    always @ (posedge pixel_clk)
    begin
        if ((phcount >= x && phcount < (x+WIDTH)) && (pvcount >= y && pvcount
< (y+HEIGHT)))
        begin
            if (image_bits != 0) pixel <= 12'h0F0;
            else pixel <= 0;
        end
        else pixel = 0;
        phcount <= hcount;
        pvcount <= vcount;
    end

    assign image_addr = (hcount - x) + (vcount - y) * WIDTH;
    start st(.addra(image_addr), .clka(pixel_clk), .douta(image_bits));

endmodule

module gen_rock
    #(parameter WIDTH = 46,
                HEIGHT = 44)
    (input pixel_clk,
    input [9:0] x, hcount,
    input [9:0] y, vcount, // x and y denote the (x,y) of the top left corner
of the image
    output reg [11:0] pixel
    );

    wire [10:0] image_addr; // retrieving color information for a pixel
    wire [3:0] image_bits; // 4 bits that encode a particular pixel
    wire [3:0] r, g, b; // 12 bits total for VGA
    reg [9:0] phcount, pvcount; // pipelined hcount and vcount

    always @ (posedge pixel_clk)
    begin
        if ((phcount >= x && phcount < (x+WIDTH)) && (pvcount >= y && pvcount
< (y+HEIGHT)))
            case (g)
                4'b1111: pixel <= 0;
                default: pixel <= {r,g,b};
            endcase
        // else pixel = DEFAULT;
        else pixel = 0;
        phcount <= hcount;
        pvcount <= vcount;
    end

    assign image_addr = (hcount - x) + (vcount - y) * WIDTH;
    rock rk(.addra(image_addr), .clka(pixel_clk), .douta(image_bits));

    color_map cm(image_bits, r, g, b); // find the corresponding rgb values
endmodule


module gen_sword
    #(parameter WIDTH = 59,
                HEIGHT = 18)
    (input pixel_clk,
    input [9:0] x, hcount,
    input [9:0] y, vcount, // x and y denote the (x,y) of the top left corner
of the image
    output reg [11:0] pixel
    );

    wire [10:0] image_addr; // retrieving color information for a pixel
    wire [3:0] image_bits; // 4 bits that encode a particular pixel
    wire [3:0] r, g, b; // 12 bits total for VGA
    reg [9:0] phcount, pvcount; // pipelined hcount and vcount

    always @ (posedge pixel_clk)
    begin
        if ((phcount >= x && phcount < (x+WIDTH)) && (pvcount >= y && pvcount
< (y+HEIGHT)))
            case (g)
                4'b1111: pixel <= 0;
                default: pixel <= {r,g,b};
            endcase
        // else pixel = DEFAULT;
        else pixel = 0;
        phcount <= hcount;
        pvcount <= vcount;
    end

    assign image_addr = (hcount - x) + (vcount - y) * WIDTH;
    sword_pic s(.addra(image_addr), .clka(pixel_clk), .douta(image_bits));

    color_map cm(image_bits, r, g, b); // find the corresponding rgb values
endmodule

module le_musique
    (
     input [5:0] addr,
     output reg [40:0] freq_and_timing
    );

    always @ (*)
    begin
        case(addr)
            6'd0:  freq_and_timing <=
41'b01011111101111011001001100010010110100000;
            6'd1:  freq_and_timing <=
41'b00000000000000000101010111010100101010000;
            6'd2:  freq_and_timing <=
41'b10000000011111101001001100010010110100000;
```

```
            6'd3:  freq_and_timing <=
41'b00000000000000000100110001001011010000000;
            6'd4:  freq_and_timing <=
41'b10011000100101101001001100010010110100000;
            6'd5:  freq_and_timing <=
41'b00000000000000000101111101011110000100000;
            6'd6:  freq_and_timing <=
41'b01101110111110010001001100010010110100000;
            6'd7:  freq_and_timing <=
41'b00000000000000000011100100111000011100000;
            6'd8:  freq_and_timing <=
41'b01100101101110011000111101000010010000000;
            6'd9:  freq_and_timing <=
41'b00000000000000000011111011110001010010000;
            6'd10: freq_and_timing <=
41'b01101100100000011001001100010010110100000;
            6'd11: freq_and_timing <=
41'b00000000000000000001110010011100001110000;
            6'd12: freq_and_timing <=
41'b01110001100011011001001100010010110100000;
            6'd13: freq_and_timing <=
41'b00000000000000000011100100111000011100000;
            6'd14: freq_and_timing <=
41'b10000000011111101001001100010010110100000;
            6'd15: freq_and_timing <=
41'b00000000000000000010011000100101101000000;
            6'd16: freq_and_timing <=
41'b01001001111110110000111101000010010000000;
            6'd17: freq_and_timing <=
41'b00000000000000000010011000100101101000000;
            6'd18: freq_and_timing <=
41'b01000000001111110000100110001001011010000;
            6'd19: freq_and_timing <=
41'b00000000000000000001110010011100001110000;
            6'd20: freq_and_timing <=
41'b00111000110001101001001100010010110100000;
            6'd21: freq_and_timing <=
41'b00000000000000000011100100111000011100000;
            6'd22: freq_and_timing <=
41'b01000101110000100000111101000010010000000;
            6'd23: freq_and_timing <=
41'b00000000000000000001110010011100001110000;
            6'd24: freq_and_timing <=
41'b01000000001111110000100110001001011010000;
            6'd25: freq_and_timing <=
41'b00000000000000001000010110000001110110000;
            6'd26: freq_and_timing <=
41'b01001001111110110000111101000010010000000;
```

```
            6'd27: freq_and_timing <=
41'b00000000000000000011100100111000011100000;
            6'd28: freq_and_timing <=
41'b01011101111001100000111101000010010000000;
            6'd29: freq_and_timing <=
41'b00000000000000000001110010011100001110000;
            6'd30: freq_and_timing <=
41'b01010100001011111000111101000010010000000;
            6'd31: freq_and_timing <=
41'b00000000000000000001110010011100001110000;
            6'd32: freq_and_timing <=
41'b01100101101110011000111101000010010000000;
            6'd33: freq_and_timing <=
41'b00000000000000000101111101011110000100000;
        endcase
    end
endmodule


module sound_gen(
    input clk,
    output reg sound);

    wire [40:0] freq_and_timing; // 41 bit wire that tells us the timing and
frequency for each note
    wire [16:0] freq_cycles; // register to store which frequency will be
played
    wire [23:0] duration_cycles; // gets the duration that the frequency will
be held for

    reg [5:0] addr = 0; // address will tell us what frequency and what
timings we are gonig to use

    reg [16:0] cycles = 0; // determine when to change to the next thing
    reg [23:0] timer = 0; // times how long that frequency goes for

    le_musique lm (.addr(addr), .freq_and_timing(freq_and_timing));

    assign {freq_cycles, duration_cycles} = freq_and_timing; // freq_cycles
has number of cycles for a frequency, duration_cycles has duration of the
frequency

    initial sound = 0; // oscillating output frequency determined by cycles

    always @ (posedge clk)
    begin
        if (freq_cycles == 0) sound <= 0; // no sound for when the frequency
is 0
```

```
        else if (cycles >= freq_cycles) begin // start the oscillation once
we've reached the necessary number of cycles
            sound <= ~sound; // make an oscillating tone
            cycles <= 0;
        end
        else cycles <= cycles + 1;

        if (timer == duration_cycles) begin // when the duration is reached
for that pitch, change pitches and timers
            timer <= 0; // reset the timer to 0
            if (addr == 33) addr <= 0; // loop back to the beginning of the
song
            else addr <= addr + 1; // look at the next sound to make and
determine how long it'll last
        end
        else timer <= timer + 1; // otherwise continue incrementing the timer
to the appropriate duration
    end


endmodule
```