6.111: Final Project Report

Amadou Bah, Mussie Demisse

December 11, 2020

Contents

1 Introduction **3**
1.1 From Old School to New School . . . 3
1.2 Setting up the Hardware . . . 3
1.3 Game Mechanics . . . 4
1.3.1 Starting the Game . . . 4
1.3.2 Game Controls . . . 5

2 A Quick Note on Collaboration **5**

3 capture_image *Mussie Demisse* **5**
3.1 Overview . . . 5
3.2 Block Diagram . . . 6
3.3 Design . . . 6

4 pos *Amadou Bah and Mussie Demisse* **6**
4.1 Overview . . . 6
4.2 Block Diagram . . . 7
4.3 Design . . . 9

5 dissect *Amadou Bah and Mussie Demisse* **9**
5.1 Overview . . . 9
5.2 Block Diagram . . . 10
5.3 Design . . . 10

6 move *Amadou Bah* **11**
6.1 Overview . . . 11
6.2 Block Diagram . . . 11
6.3 Design . . . 13
6.3.1 Triggering Moves . . . 13
6.3.2 Identifying Valid Moves . . . 13
6.3.3 Performing Segment Exchange . . . 13

7 shuffle *Mussie Demisse* **14**
7.1 Overview . . . 14
7.2 Block Diagram . . . 14
7.3 Design . . . 15

8 draw *Amadou Bah* **15**
8.1 Overview . . . 15
8.2 Block Diagram . . . 15
8.3 Design . . . 16

9 menu_blob *Mussie Demisse* **17**
9.1 Overview . . . 17
9.2 Block Diagram . . . 17
9.3 Design . . . 18

10 grid_shuffle_game *Amadou Bah and Mussie Demisse* **18**
10.1 Overview . . . 18
10.2 Block Diagram . . . 18
10.3 Design . . . 19
10.3.1 Managing Two Instances . . . 19
10.3.2 Communicating With top_level . . . 19
10.3.3 Displaying Two Boards . . . 19

11 ones_display/tens_display *Amadou Bah* **20**
11.1 Overview . . . 20
11.2 Block Diagram . . . 20
11.3 Design . . . 21

12 progress-bar *Mussie Demisse* **21**
12.1 Overview . . . 21
12.2 Block Diagram . . . 21
12.3 Design . . . 21

13 win_blob and lose_blob *Mussie Demisse* **22**
13.1 Overview . . . 22
13.2 Block Diagram . . . 23
13.3 Design . . . 23

14 top_level *Amadou Bah and Mussie Demisse* **23**
14.1 Overview . . . 23
14.1.1 Home of the Main Modules . . . 24
14.1.2 Home of the Game FSM . . . 24
14.2 Block Diagram . . . 24
14.3 Design . . . 25
14.3.1 Game FSM Logic . . . 25
14.3.2 Button Pressed vs Button Released . . . 25
14.3.3 Display Logic . . . 26

15 Challenges **26**
15.1 Timing Challenged . . . 26
15.2 Running out of BRAM . . . 26

16 Possible Improvements / future prospects **26**

17 Appendix **27**
17.1 Code . . . 27
17.2 Large Block Diagrams . . . 27

1 Introduction

Our 6.111 final project brings back the grid shuffle game. Traditional versions of this game work in the following way:

1. Players choose one image from a built-in array of pictures. Along with that, they specify the level of difficulty by selecting a choice of grid size. Typically 3x3, 4x4, and 5x5.

2. That image is then segmented into a square grid of the selected size and a single cell has been removed, leaving behind a hole; typically, one of the corner pieces is removed since doing so does not jeopardise the main features of the overall image.

3. Finally, the segmented image is then shuffled, and the player is tasked with reconstructing the image via a sequence of **valid moves**, where a **valid move** involves swapping the positions of a single cell that is adjacent (excluding diagonals) to the hole with that of the hole.

From there, the player keeps shifting pieces around on the game board until they successfully place all the square ”tiles” in their correct position, to reconstruct the original image.

1.1 From Old School to New School

Unlike the traditional grid shuffle game, however, where the player is constrained to the built-in images, in our version, the player decides what images they would like to use (`game_image`). This freedom to decide comes from the fact that the player has access to an on-board camera. Additionally, the player gets to choose between four different game modes:

- CLASSIC: This is akin to the traditional version described above. In this game mode, unless the player gives up they do not lose. They win by completing the reconstruction.

- TWO PLAYER: In this mode, the player can invite a friend to play along. They will each have their own game board (and game ”controller”) to reconstruct simultaneously. For the sake of fairness, however, the two game boards are shuffled the exact same way, so they have the same initial state. The player that reconstructs their image the fastest wins.

- TIME TRIAL: In this mode, the player races against the clock. The allotted time depends on the number of moves it took to shuffle the image, and there is a progress bar that visually expresses how long is left. If the player does not reconstruct the image in the allotted time, they lose.

- MAX MOVES TRIAL: In this mode, the player is only allowed to make a limited number of moves. The maximum allowed number of moves depends on the number of moves it took to shuffle the image, and there is a seven segment display below the game board showing the number of moves left.

Even so, like old-school grid shuffle, the players still have the option of further challenging themselves by choosing from the available game sizes: 3x3, 4x4, or 5x5.

1.2 Setting up the Hardware

The hardware is composed of the FPGA, a screen, a camera module attached to an ESP8266, an arcade joystick, with four resistors and some wires.

- For information/instructions on assembling the camera module, please follow this link `https://jodalyst.com/6111/camera_info/`

- The FPGA outputs visual information through a VGA port, so you will need a VGA-to-HDMI connector depending on your display inputs.
- The arcade analog stick has five wires: The wire furthest from the head of the analog stick connects to GND on the JC PMOD ports. The next four wires can be plugged into the JC 0,1,2 and 3 ports with a 2K ohm between each connection except for the ground.

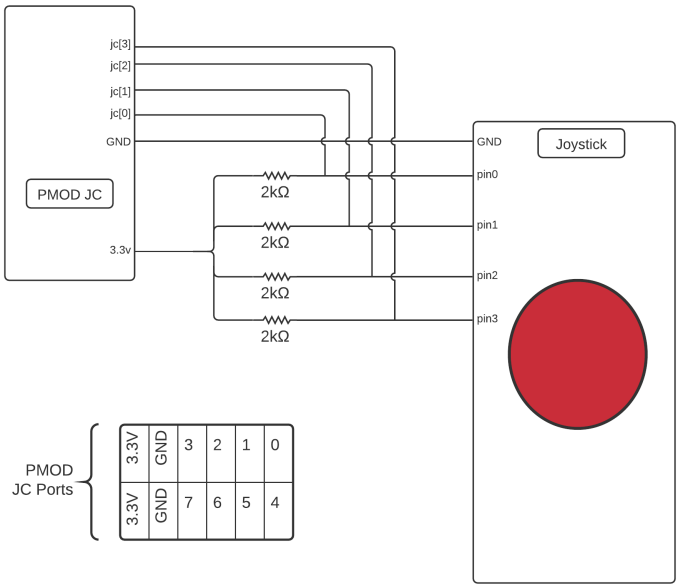

Figure 1: The circuit diagram for connecting the Joystick to the FPGA.

1.3 Game Mechanics

1.3.1 Starting the Game

The game begins on the main menu, where the player can use `btnu` and `btnd` on the FPGA to choose between each of the game modes. At the same time, the player can use `btnl` and `btnr` to chose their desired game board size. Once they are satisfied with their choice of game mode and size, they can use `btnc` to lock their selections. At which point, the camera feed will be shown. With it, the player orients the camera to take a photo of their choice, and uses `btnc` again to capture and save the image. Note that the joystick cannot be used in the main menu.

The captured image, which we will refer to as the `game_image`, is then dissected by the system into the specified grid size. After that, the program creates the `hole` by removing the bottom-right square; we achieve this effect by never showing the pixels on the grid identified as the `hole`. From there, the objective is the same: reconstruct the image following a sequence of **valid moves** to successfully constructs the `game_image`.

Upon successful reconstruction in a single player mode, the game will display a congratulatory image. If the player loses the game (in the modes where losing is possible), we show a sad image that prompts them to try again. In TWO-PLAYER mode, the same way there will be two small game board for each player, two images will be displayed once a successful reconstruction is achieved: The congratulatory image on the side of the winner, and the sad face on the side of the loser.

1.3.2 Game Controls

Players can interact with the game using either the FPGA buttons and switches, or using the joystick. While some controls have already been discussed in the section above, the following is a more comprehensive list:

- `btnu` : On the main menu, this button will move the **mode** selection icon upward. Likewise, in TWO-PLAYER mode, it will allow the second player to move the hole upward.
- `btnd` : On the main menu, this button will move the **mode** selection icon downward. Likewise, in TWO-PLAYER mode, it will allow the second player to move the hole downward.
- `btnl` : On the main menu, this button will move the **size** selection icon to the left. Likewise, in TWO-PLAYER mode, it will allow the second player to move the hole to the left.
- `btnr` : On the main menu, this button will move the **size** selection icon to the right. Likewise, in TWO-PLAYER mode, it will allow the second player to move the hole to the right.
- `btnc` : On the main menu, this button will allow the user to lock their choice of **mode** and **size**. On the Camera screen, it is used to capture and save the `game_image`.
- `sw[2]` : While inside any of the game modes, this switch allows the user to toggle between the shuffled game and the `game_image`. This is useful if the user cannot identify a ”tile’s” true position. Note that in TIME TRIAL mode, viewing the `game_image` will not pause the game.
- `sw[7]` : This is the reset switch for the system. Within any of the game modes, this switch allows the player to return to the main menu, regardless of whether reconstruction was complete.
- Analogue Stick : Only used by player one, this allows the player to move ”tiles” on the game board during reconstruction.

2 A Quick Note on Collaboration

Before delving into the module descriptions, we wanted to mention that throughout this project we used pair programming, because we wanted more experience with the practice and it was convenient. We enjoyed this process, but we feel we must state to the teaching staff that a huge majority of the work produced was produced together and even though we can split some modules up based on who was driving (typing), who debugged longer, or some other smaller factors, for some modules neither of us can claim it in good conscience, and so we are both claiming those modules.

3 capture_image

3.1 Overview

Once the player has chosen a game `mode` and `size`, they will see the output of the camera on the screen and when they click on the center button on the FPGA, the `capture_image` module will store the image that is on the screen onto BRAM, so that it can be used as the `game_image`.

3.2 Block Diagram

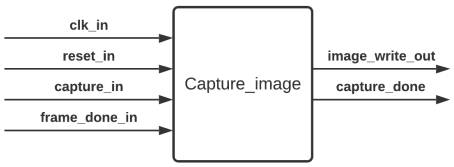

Figure 2: The capture_image module.

Inputs:

- `clk_in`: The 65Mhz clock
- `reset_in`: The reset for the system
- `frame_done_in`: if asserted, the next pixel will be a new frame

Outputs:

- `capture_done`: asserted when a full frame has been saved to BRAM.
- `image_write_out`: asserted to write to BRAM.

3.3 Design

When `capture_in` is asserted, the module waits for `frame_done_in` to be asserted before it sets `image_write_out` to HIGH. `image_write_out` is the write enable pin for the BRAM. The module sets `image_write_out` back to LOW, when `frame_done_in` is asserted again. This ensures that a full frame of the image is written to BRAM. Since the image is used by other modules, `capture_done` is used to alert them that the image has finished writing. Both `frame_done_in` and `capture_done` are impulses.

4 pos

4.1 Overview

So far, the process of reconstruction has been described as moving ”tiles” on the game board until they are in their correct position. The idea of a ”tile” will now be formalized. Consider Figure 3 below; [A] shows a shuffled game board (including the `hole`), and each of the squares on the board (which were called ”tiles”) is composed of two game elements that can be seen in [B], a view of the board from a more 3-dimensional perspective. The two game elements are: the `pos`, in green, and the `seg`.

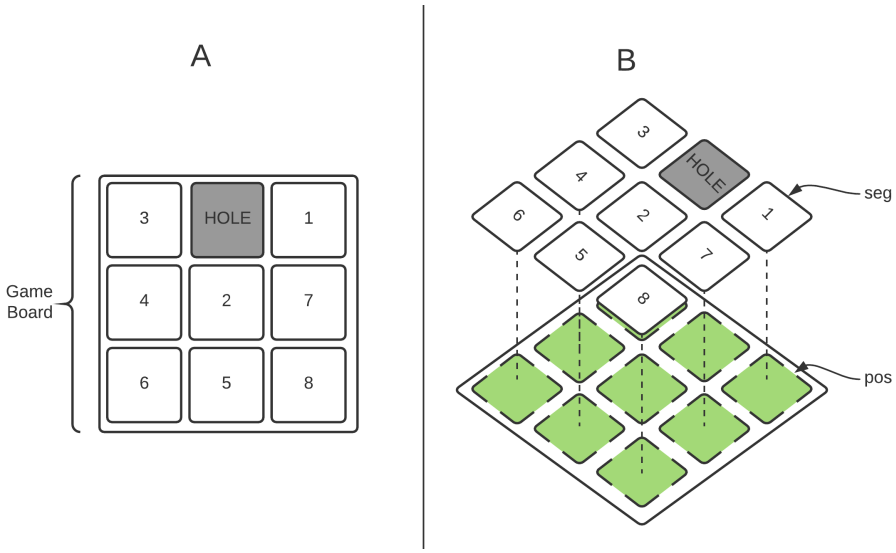

Figure 3: The left image shows the game board in a 2-dimensional space, whereas the right image is a view from a 3-dimensional perspective with the `seg` extracted from their `pos` objects.

A `seg` (short for segment) is an internal representation of a piece of the `game_image` after it has been dissected, and each `seg` element is identified by the coordinates of it's top-left corner on the captured `game_image`. These, now individual, segments are placed inside `pos` objects, though they are free to move between `pos` (this process will be explained later). On the other hand, the `pos` is the basic building block of the game, and is fixed onto the game board. Like the `seg` element, each `pos` object can be identified by the unique x and y coordinate of it's top-left corner on the game board. In addition to the holding the `seg` elements, the `pos` objects know their dimensions, and can identify whether the `seg` it holds is a `hole`.

The `pos` objects were introduced to handle the anticipated **size** selection feature. Figure 4 below shows the arrangement of all the `pos` objects, along with their indices, on the game board. The specified game **size** dictates which of those objects are activated. A 3x3 would activate the green `pos` objects for instance, so that `pos` objects outside the specified **size** do not hold any `seg` elements, and thus are not visible. While this arrangement makes **size** selection feasible, it also facilitates movement, as will be discussed later in the `move` module.

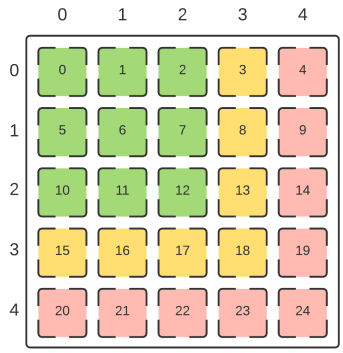

Figure 4: The arragement of the `pos` objects on the game board.

4.2 Block Diagram

Inside any of the game modes (CLASSIC, TWO-PLAYER, etc), the `pos` object communicates with other modules in the game, letting them know what pixels to show. The block diagram below shows the inputs and outputs of the `pos` module:

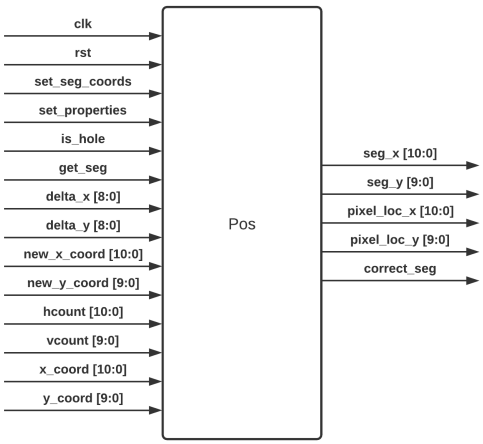

Figure 5: The pos module.

Inputs:

- `clk`: The 65Mhz clock
- `rst_in`: The reset for the system
- `set_seg_coords`: When asserted, the `pos` assumes the `seg` whose coordinates are specified by `new_x_coord` and `new_y_coord`.
- `set_properties`: When asserted, the coordinates of the `pos` as well as those of the `seg` are updated to `x_coord` and `y_coord`. This only happens once.
- `is_hole`: When asserted, informs the `pos` that the `seg` it holds is a `hole`.
- `get_seg`: Determines whether the `pos` should return the coordinates of it's `seg` via the outputs: `seg_x` and `seg_y`.
- `delta_x[8:0]`: The width of a `pos`
- `delta_y[8:0]`: The height of a `pos`
- `new_x_coord[10:0]`: Used to set the `seg` x coordinate
- `new_y_coord[9:0]`: Used to set the `seg` y coordinate
- `x_coord[10:0]`: Used to set the `pos` and `seg` x coordinate
- `y_coord[9:0]`: Used to set the `pos` and `seg` y coordinate
- `hcount[10:0]`: The x coordinate of where a pixel is to be drawn on the display. Ranges from 0 to 1023
- `vcount[9:0]`: The y coordinate of where a pixel is to be drawn on the display. Ranges from 0 to 767

Outputs:

- `seg_x[10:0]`: The `seg` x coordinate if `get_seg` is asserted, 0 otherwise
- `seg_y[9:0]`: The `seg` y coordinate if `get_seg` is asserted, 0 otherwise

- pixel_loc_x[10:0]: For a given hcount, this indicates where the pixel corresponding to the original image would be if it is within the pos, 0 if it's not
- pixel_loc_y[9:0]: For a given vcount, this indicates where the pixel corresponding to the original image would be if it is within the pos, 0 if it's not
- correct_seg: An indication of whether the pos is holding the correct seg.

4.3 Design

At the start of a new game, the pos starts off not having it's unique x and y coordinates, and it also does not hold a seg. However, when set_properties is asserted, the values of x_coord and y_coord are set as the unique coordinates of the pos as well as those for the seg. Though movement will be described in detail later, the main point right now is that when a move is being performed set_seg_coords is asserted, which tells the pos object to assume the seg element whose coordinates are given by new_x_coord, new_y_coord.

This module also plays a part in determining if a correct reconstruction is achieved. The value of correct_seg is HIGH if the coordinates of the pos matches that of the seg. Along those lines, if every pos object asserts it's correct_seg, then reconstruction is complete and the player has won. Intuitively, this makes sense because it implies that the pos is holding on to the correct segment from the original image. Another way to make sense of this is to remember that when set_properties is asserted, seg and pos have the same value (before any shuffling).

Finally to identify the pixel that must be displayed on the screen during any of the game modes, pos relies on the values of hcount and vcount. If both values are within bounds of the pos, that is:

pos_x<hcount && hcount<pos_x+delta_x
pos_y<vcount && vcount<pos_y+delta_y

where pos_x and pos_y are, respectfully, the x and y coordinates unique to the pos object, then considering the seg it holds, the pos object calculates the relative location of the pixel on the original game_image using the following transformation:

pixel_loc_x = hcount - pos_x_coord + seg_x_coord
pixel_loc_y = vcount - pos_y_coord + seg_y_coord.

If however, the values of hcount and vcount are not within bounds, the output, pixel_loc_x and pixel_loc_x, are 0. Notice, how hcount and vcount cannot be within the boundaries two pos objects at the same time. This is important for drawing the grid on the game board, as will be seen later.

5 dissect

5.1 Overview

Once the player captures and saves the game_image to BRAM, the image must then be transformed into a pos-seg representation. This means assigning unique coordinates to the pos objects along with their seg elements. The role of this module is to assign those initial coordinates, and determine the dimensions of pos and seg.

5.2 Block Diagram

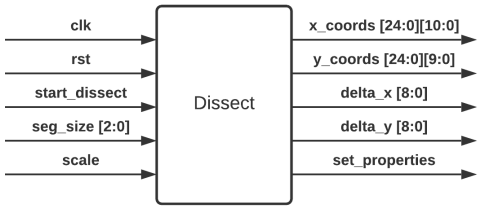

Figure 6: The dissect module.

Inputs:

- `clk_in`: The 65Mhz clock
- `rst_in`: The reset for the system
- `start_dissect`: When asserted, the module starts calculating dimensions before assigning coordinates
- `seg_size[2:0]`: Is either 3'b011, 3'b100, or 3'b101, and used to calculate dimensions and coordinates
- `scale`: Asserted if the game board size is 640x480, otherwise 320x240

Outputs:

- `x_coords[24:0][10:0]`: This 25 by 11 pixel two dimensional array outputs the x coordinate value for the 25 `pos` objects.
- `y_coords[24:0][9:0]`: This 25 by 10 pixel two dimensional array outputs the y coordinate value for the 25 `pos` objects.
- `delta_x[8:0]`: The width of a `pos` object. and represents the horizontal distance between two, side-by-side `pos` objects.
- `delta_y[8:0]`: The height of the `pos` object, and represents the vertical distance between two `pos` objects.
- `set_properties`: It marks the end of dissection, when `pos` objects and `seg` elements are assigned their coordinates and dimensions

5.3 Design

The width of a `pos` is calculated by dividing the width of the screen by the `seg_size`, and then doubling it if `scale` is asserted. This same process holds for the `pos` height, except the screen height is divided. The x coordinates of the 25 `pos` objects are obtained by multiplying their width by their column value in Figure 4, and their y coordinates by multiplying their height by the row value.

6 move

6.1 Overview

Because the main objective in this project is reconstructing a shuffled image, movement is a necessary component. From any given game board state, the idea is to replace the position of the hole with that of the one of its neighbors. Recall, we define a **valid move** as a direction (using the `hole` as a reference point) that would allow the `hole` to successfully exchange positions with the neighbor in the specified direction. So, one of the roles of this module is ensure that moves are valid.

Still, in order to gauge a move's validity, the module must first know which `pos` contains the `hole` at all times, and, second, be able to identify its neighboring `pos` objects. Knowing this ensures that the module can correctly decide whether a neighbor in a specified direction is part of the game board. Finally, the game must correctly perform the position exchange, assuming that a **valid move** is requested.

That said, at a high level, the movement module is a large FSM that transitions between three states:

1. IDLE: When the user is idle and not specifying any moves, this module remains in this state. Also, the module returns to this state whenever a move is complete to wait for new move instructions.
2. CHECK_VALID: in this state, the module assesses the specified direction, to determine if the associated neighbour is part of the game board. The module returns to the IDLE state if a move is not possible, and proceeds to the PERFORM_MOVE state otherwise.
3. PERFORM_MOVE: here, the module swaps the hole and the neighbour in the specified direction, to complete a **valid move**. Then returns to the IDLE state.

6.2 Block Diagram

The movement module plays a big role for the functionality of the game, and number of inputs and outputs it handles reflects that. Below is a block diagram for `move`:

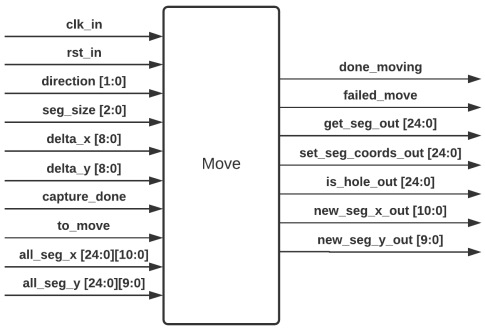

Figure 7: The move module

Inputs:

- `clk_in`: The 65Mhz clock
- `rst_in`: The reset for the system
- `direction[1:0]`: This 2-bit value is a representation of the specified direction, such that `2'b11` is UP, `2'b00` is DOWN, `2'b01` is RIGHT, and `2'b10` is LEFT

- `seg_size[2:0]`: The size of the game board. Since the game offers 3 size options for each of the game modes, a `3'b011` value of `seg_size` would indicate a 3x3 game board, a `3'b100` value would indicate a 4x4 board, and `3'b101`, a 5x5

- `delta_x[10:0]`: The width, in pixels, of one grid cell on the game board. Because the camera produces a 320x240 image resolution, we also store it in BRAM along those dimensions. The value of `delta_x` varies depending on `seg_size`, and game `mode`. For the available board sizes in two player mode, 3x3, 4x4, and 5x5, the values for `delta_x` are `11'd107`, `11'd80`, and `11'd64` respectfully. For all the other modes the previous values are just doubled.

- `delta_y[9:0]`: The height, in pixels, of one grid cell on the game board. Because the camera produces a 320x240 image resolution, we also store it in BRAM along those dimensions. So the value of `delta_y` also varies depending on `seg_size`, and game `mode`. For the available board sizes in two player mode, 3x3, 4x4, and 5x5, the values for `delta_y` are `10'd80`, `10'd60`, and `10'd48` respectfully. For all the other modes the previous values are just doubled.

- `capture_done`: An indicator that a picture, to use as the game image, has been taken and saved to BRAM.

- `to_move`: An indicator that a move should be attempted, given the current `direction`. This value, for instance, is **high** if a player presses one of the direction buttons on the FPGA (for two-player mode), or moves the analogue stick, and the value returns to **low** once the button/analogue stick is released.

- `all_seg_x[24:0][10:0]`: this packed array holds the 11-bit x-coordinates of the `seg` contained within each `pos` object. Each `pos` object, whose index on the board (as shown earlier) matches the index in the array, will populate an index of `all_seg_x` corresponding to it's position with either a value of `0`, or a the x-coordinate of it's `seg`. That value will depend on an input to the `pos` module dictated by an output, `get_seg_out[24:0]`, of this module. More on `get_seg_out[24:0]` later.

- `all_seg_y[24:0][9:0]`: this input is similar to `all_seg_x[24:0][10:0]`, except it holds the y-coordinates.

Outputs:

- `done_moving`: An indicator that a move was successfully completed

- `failed_move`: An indicator that a move could not be completed.

- `get_seg_out[24:0]`: The index of each bit in this 25-bit number represents the index of each of the 25 `pos` objects on the game board, and the value of the bit (`0/1`) indicates whether the `pos` at that index should populate `all_seg_x[24:0][10:0]` and `all_seg_y[24:0][9:0]` with the x and y coordinates of it's `seg` or with a value of `0`. At any given point, only one of the 25 bits will have a value of `1`. This is because, to perform a move, the module only needs the `seg` of the `pos` that will exchange `seg` values with the `pos` containing the hole.

- `set_seg_coords_out[24:0]`: Like `get_seg_out[24:0]`, the index of each bit in this 25-bit number represents the index of each of the 25 `pos` objects on the game board. The value (`0/1`) indicates whether that `pos` should accept new `seg` coordinates following a move.

- `is_hole_out[24:0]`: The index of each bit in this 25-bit number represents the index of each of the 25 `pos` objects on the game board. The value (`0/1`) indicates whether that `pos` is a `hole`. Since the board can only have one `hole`, only a single bit in `is_hole_out[24:0]` can be **high** at any given point.

- new_seg_x[10:0]: The x-coordinates of the seg that a pos can use to replace it's current seg x-coordinate value. A pos only assumes this value if it's index in set_seg_coords_out[24:0] is **high**.
- new_seg_y[9:0]: Same as new_seg_x[10:0], except this is the for the y-coordinate.

6.3 Design

6.3.1 Triggering Moves

As mentioned before, the system offers two ways of accepting movement instructions from the player(s). For instance, in the two player mode, move instructions are accepted from both the arcade joystick and the direction buttons on the FPGA; whereas in all of the single player modes, only movements specified from the joystick are accepted.

6.3.2 Identifying Valid Moves

So far, a move has been described as exchanging the position of the hole with that of a neighbour in the specified direction. However, while that description makes sense visually, it is not exactly what happens under the hood. Recall, pos objects are defined as fixed positions on the game board, each holding the coordinates of a specific seg of the dissected image captured using the camera. If two positions on the game board swap the portion of the game image they hold, the player gets a visual sense that a move was made. So, along those line, a move means simply exchanging the seg coordinates that two neighbouring pos objects hold.

Due to the nature of the game, a move will always involve the hole. So, when a game starts (mode/size chosen, image captured and dissected), the move module stores the coordinates of the pos object containing the hole along with it's index on the board, and the coordinates of the seg, from the captured image, associated with that pos. With that, once the signal to make a move arrives via to_move, depending on the direction, the module will add or subtract delta_x and delta_y from the stored hole_pos coordinates. The result will be x and y coordinates of the neighbour whose seg coordinates are to be exchanged with those of the hole_pos. Now given the limitation of the game board, the x and y coordinates of a pos must be less than the dimensions of the captured image (320x240). So, following the addition/subtraction, if the resulting value has a potential to become negative, or is $\geq$ the boundary dimensions, then the move is invalid, and failed_move is asserted. Otherwise, the move is valid, and the module proceeds to performing the segment exchange.

6.3.3 Performing Segment Exchange

Assuming the move was valid, the module begin segment exchange by manipulating the value at the index of the associated with the hole's neighbour (call it neighbour_index) in get_seg_out[24:0], set_seg_coords_out[24:0], is_hole_out[24:0]. Given the index of the pos containing the hole (call it hole_index), it is possible to calculate the index of the neighbour in the specified direction[1:0] by taking advantage of the board's layout as follows:

1. LEFT: subtract 1 from the index of the hole.
2. RIGHT: add 1 to the index of the hole.
3. UP: subtract 5 from the index of the .
4. DOWN: add 5 to the index of the hole.

The above was a good insight, from Mussie, that helps the module to avoid making expensive multiplications in order convert coordinates to indices.

Once the index is identified, the module first sets the value of `get_seg_out[neighbour_index]` to `1`, while keeping all other bits `0`. This signals the `pos` with that index to provide it's `seg` coordinates via `all_seg_x[24:0][10:0]` and `all_seg_y[24:0][10:0]`, so that the values of `new_seg_x[10:0]` will be assigned the value of x-coordinates of the `seg` contained by the neighbouring `pos`. A similar idea for the y-coordinates. The module will then assert the value of `set_seg_coords_out[hole_index]`, while keeping all other bits `0`, to signal the `pos` currently containing the `hole`, to assume the new x and y segment coordinates. A similar process happens simultaneously to get the neighbour `pos` to assume the `seg` coordinates associated with the `hole`. Finally, the values in `is_hole_out[24:0]` are updated (`is_hole_out[hole_index]=0` and `is_hole_out[neighbour_index]=1`) to reflect the fact that the hole index has changed.

7 shuffle

7.1 Overview

Once the player has taken a picture and the `dissect` module has assigned all the `pos`es with their fixed x,y coordinates and their mutable segment x,y coordinates, the image segments must be shuffled on the game board, as is shown below.

7.2 Block Diagram

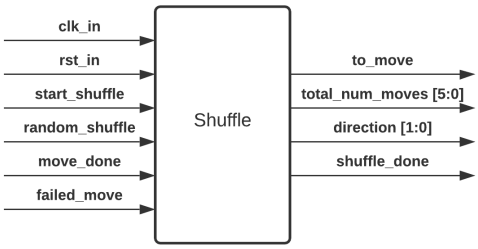

Figure 8: The shuffle module.

Inputs:

- `clk_in`: The 65Mhz clock
- `rst_in`: The reset for the system
- `start_shuffle`: if asserted initiates shuffle FSM only if one hasn't already started
- `random_val`: 1 bit random value from the pseudo-random crc generator
- `move_done`: asserted when a valid move has been completed
- `failed_move`: asserted when a in-valid move has failed to complete

Outputs:

- `progress_bar_pixel[11:0]`: The 12 bit RGB value to be drawn at a given `hcount_in` and `vcount_in`

7.3 Design

The shuffled state is accomplished by ordering a pseudo-random number of pseudo-random moves. `start_shuffle` initiates a finite state machine (FSM). The FSM consists of getting the number of moves, getting the directions for those moves, and ordering those moves sequentially.
`start_shuffle` being asserted transitions the FSM to the `get_num_moves` state.

1. GET_NUM_MOVES: In this state `random_val` is added to decreasing indices of the `num_moves` array for five clock cycles. The number of moves is constrained by localparams `max_allowed_moves` and `min_num_moves`. After the fifth clock cycle the FSM will transition to the `get_directions` state.

2. GET_DIRECTIONS: In this state `random_val` will fill the `directions` array. Within this state it is not possible to have two consecutive moves that cancel each other out (i.e right then left). The representation of directions is in such a way that two opposing directions will add up to give us 2'b11. So before we add a value to the array, we check if the sum of the last two bits of the directions array and the direction made by the combination of the last bit of the `directions` array and the new `random_val` sum up to 2'b11, if it does then `random_val` is flipped before being added to the `directions` array. Once we have filled up the `directions` array up to `num_moves` we will transition to the `send_directions` state.

3. SEND_DIRECTIONS: The `send_directions` state sets `to_move` and `direction`, both of which are inputs to the `move` module. A new move order is sent when it's the first move or a valid move was completed (`move_done`) or if the previous move was invalid (`failed_move`). A new move is sent after an invalid one so as not to stall the process. An invalid move is not added to the `total_num_moves` counter that will be used to decide how many moves the player will be allowed to make in the max moves trial game mode. Once all the moves are sent, we set `shuffle_done` to HIGH.

8 draw

8.1 Overview

Reconstruction can only be attempted if the player can see the shuffled grid. This module is responsible for deciding what pixels of the captured image to show on the display as the player attempts to piece the shuffled image the back together. Recall, every Pos module holds a segment of the original image, and knows if the values of `hcount` and `vcount` are with it's boundaries; so at any point the `pos` object can identify the x and y pixel coordinates associated with `hcount` and `vcount` within `seg` that it holds. The Draw module accepts those coordinates from all 25 `pos` objects to identify the BRAM address for the pixel at the current `hcount` and `vcount`.

8.2 Block Diagram

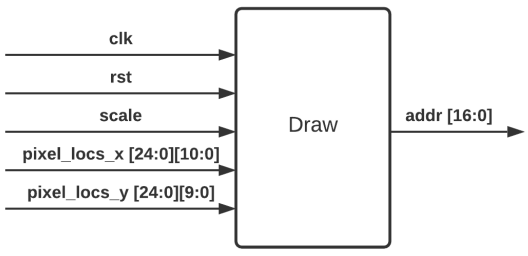

Figure 9: The draw module.

Inputs:

- `clk_in`: The 65Mhz clock
- `rst`: The system reset
- `scale`: An indicator of whether the game board should be drawn larger. For instance, apart from two-player mode where the game board needs to be restricted to a 320x240 so that two boards can be displayed at once, in the other single-player modes the game board can be scaled up to 640x480 for a better playing experience. So if `scale` is asserted, then then the game is scaled.
- `pixel_locs_x[24:0][10:0]`: This packed array holds 11-bit numbers that represent the current `hcount` position relative to the segments of the original image.
- `pixel_locs_y[24:0][9:0]`: This packed array holds 11-bit numbers that represent the current `vcount` position relative to the segments of the original image.

Outputs:

- `addr[16:0]`: The BRAM address for the pixel on the captured image at the current `hcount` and `vcount`.

8.3 Design

As hinted above, this module takes advantage of the work done in other modules, specially the `pos` module, to decide the BRAM address from which to retrieve pixels. Each of the Pos modules provide the coordinates of a pixel through in the input parameters `pixel_locs_x[24:0][10:0]`, and `pixel_locs_y[24:0][9:0]`. The coordinates of `hcount` and `vcount` can only be contained with one `pos` object, which means only that `pos` object will populate the it's index in `pixel_locs_x[24:0][10:0]` and `pixel_locs_y[24:0][9:0]` with a non-zero number.

To identify the address, the Draw module first sums up the values in `pixel_locs_x[24:0][10:0]` and `pixel_locs_y[24:0][9:0]`, to obtain x and y-coordinates of interest as follows:

```
pixel_x = pixel_locs_x[0]+pixel_locs_x[1]+...
pixel_y = pixel_locs_y[0]+pixel_locs_y[1]+...
```

keeping in mind that only the `pos` module whose boundaries contain `hcount` and `vcount` will contribute to the sum. Finally the BRAM address is obtained by:

```
addr = pixel_x + pixel_y*image_width,
```

where `image_width` is 320. Still, if scale is asserted, the same pixel is used twice to provide a larger display. This doubling is achieved by:

```
addr = pixel_x>>1 + (pixel_y>>1)*image_width.
```

9 menu_blob

9.1 Overview

Like any good game with several options the game needs a menu. The player gets to choose between 4 game modes (classic, two player, time trial, and max move trial) and 3 game sizes (3x3, 4x4, and 5x5). This module is strictly for visual display purposes and doesn't manage any of the game modes or game sizes. In order to show the current selections for both the options we use two red circles. One red circle can be found to the left of the current game mode selection and another red circle can be found below the current game size selection.

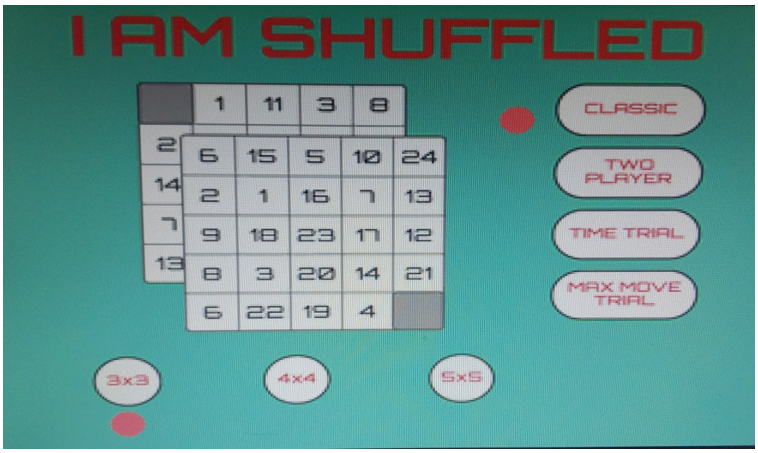

Figure 10: An image of the menu.

9.2 Block Diagram

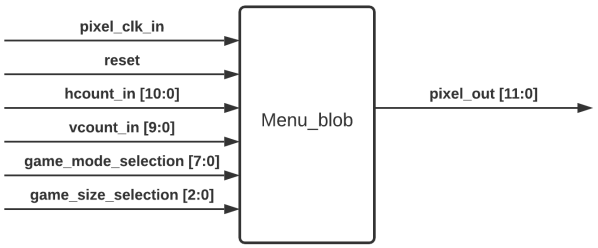

Figure 11: The menu_blob module.

Inputs:

- `clk_in`: The 65Mhz clock
- `hcount_in[10:0]`, `vcount_in[9:0]`: where we are drawing currently
- `game_mode_selection[7:0]`: represents which game mode selection is currently selected or where the game mode selector module (red circle) should be
- `game_size_selection[2:0]`: represents which game size selection is currently selected or where the game size selector module (red circle) should be

Outputs:

- `pixel_out[11:0]`: 12 bit RGB value to be drawn at a given `hcount_in` and `vcount_in`

9.3 Design

Since this is a display module it's output is a 12 bit RGB pixel value `pixel_out`, and it's inputs include `hcount_in` and `vcount_in`. The menu image was imported into a rom, along with coe files for the red, green, and blue values, all of which was generated using the supplied python `image_to_coe.py` script. The 320x240 image is up-sampled to show the menu image at 640x480. Inside the `menu_blob` module there are two instances of the selector module. The selector module is a display module for the red circle that is used to highlight the player's current selected option. `game_mode_selection` and `game_size_selection` are used in two switch-cases to dictate where the red circles will be drawn.

10 grid_shuffle_game

10.1 Overview

Since the ultimate goal was to implement a two player version, the game was designed to allows several instances of the game to share information with each other, such as using the same `game_image`, and having the same sequence of moves for shuffling. These game instances are objects of `grid_shuffle_game`. This module brings together many other modules, and also communicates with `top_level`. For instance the `dissect`, `move`, `draw`, and 25 `pos` objects all live here.

10.2 Block Diagram

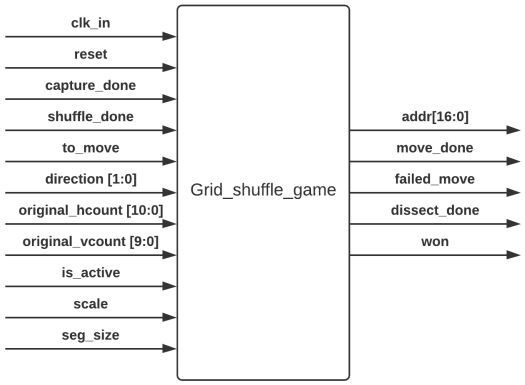

Figure 12: The grid_shuffle_game module.

Inputs:

- `clk_in`: The 65Mhz clock
- `reset`: The reset for the system
- `capture_done`: When asserted, signifies frame has been written to BRAM, and dissect can now start
- `shuffle_done`: Keeps the game instance from claiming victory before the player has moved, such as right after dissect is done
- `to_move`: Used by the player's controls and the `shuffle` module to controls the `move` module
- `direction[1:0]`: Used by the player's controls and the `shuffle` module to specify a direction for the `move` module to use

- original_hcount[10:0], original_vcount[9:0]: The x and y coordinates of where a pixel is to be drawn on the display
- is_active: Decides if this instance is actually used or not
- scale: Used to decide the size of the game board
- seg_size[2:0]: Represents the game **size** selection

Outputs:

- addr[16:0]: Address of BRAM containing game_image to read from.
- move_done: Asserted when a move is completed successfully.
- failed_move: Asserted when an attempt to make a move fails.
- dissect_done: Asserted when dissect is done, so that shuffle may begin.
- won: Asserted when all the pos objects hold the correct seg element.

10.3 Design

10.3.1 Managing Two Instances

Though there are two instances of this module, to handle TWO-PLAYER mode, the outputs of each instance is only relevant when is_active is asserted, otherwise the module outputs dummy data such as asserted values of move_done and dissect_done so that they don't block other processes in top_level.

10.3.2 Communicating With top_level

In addition to the is_active command, this module receives several queues from top_level at different points in the game. After a game_image is captured, the game board must be shuffled before the player can make movements. The input capture_done, informs the module when a new game_image is saved in BRAM so that the grid_shuffle_game know to begin dissecting that image, before asserting dissect_done. At that point, the top_level starts shuffling the game boards, and asserts shuffle_done, which allows this module to start accepting player-specified moves. That said, when the player attempts to make a move, to_move is asserted, and direction is specified in top_level. The grid_shuffle_game accepts these instructions and passes them to the move module, and depending on the nature of the attempted move, the grid_shuffle_game will either assert failed_move or dissect_done. Lastly, the two instance use won to specify when a winning condition is met.

10.3.3 Displaying Two Boards

Drawing the game boards depends on hcount and vcount, but in TWO-PLAYER mode the game boards should not overlap even though they read from the same BRAM. To handle this the grid_shuffle_game module has the x_offset parameter that represents where the left most pixel of the game instance should be. For the first game instance x_offset is zero, but for the second instance x_offset is 370, because 320 pixels are used by the first, and 50 pixels are used as a border. This offset is necessary for recalculating the relative hcount and vcount that will be passed on to other modules, which can then identify the address, addr[15:0], of the pixel to use from BRAM. This is why there can be two instances that do not overlap but can still communicate.

11 ones_display/tens_display

11.1 Overview

Instead of relying on the 7-segment displays on the FPGA, immediately under the game board in MAX MOVE TRIAL, the game shows the number of moves the player is allowed to used for reconstruction. This number decreases with each `valid_move`. Two modules `tens_display` and `ones_display` work in tandem to accomplish this visual:

Figure 13: The number of moves allowed as shown on the screen in MAX MOVE TRIAL mode.

11.2 Block Diagram

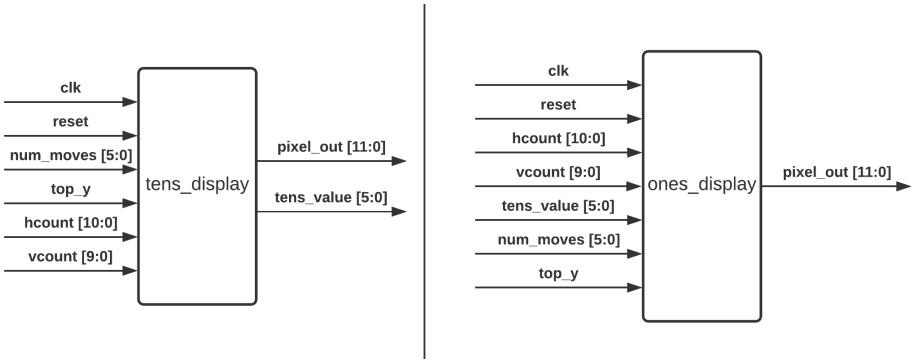

Figure 14: The tens_display and ones_display modules.

Inputs:

- `clock`: The 65Mhz clock
- `reset`: The reset for the system
- `hcount[10:0]`, `vcount[9:0]`: where we are drawing currently
- `num_moves[5:0]`: The number of moves the player is allowed to make
- `top_y`: Indicates where the displays should be drawn
- `tens_value[5:0]`: The value in the tens position of `num_moves[5:0]`

Outputs:

- `tens_value[5:0]`: Same as in the input
- `pixel_out[11:0]`: The 12 bit RGB value that we want to draw at a given `hcount` and `vcount`

11.3 Design

At first glance, it is clear that appearance of this countdown was inspired from the 7-segment LEDs on the FPGA. As the names suggest, given a value of `num_moves`, say `27`, the `tens_display` module is responsible for displying the tens value (`2`), and the `ones_display` module will show the ones value. First, the `num_moves[5:0]` is passed into `tens_module` which identifies the tens value of the number. This is possible because the game has a cap on the maximum number of moves used to shuffle (`40`). Once the tens value is identified, it is multiplied by `10` and passed into the ones `ones_display` module via `tens_value`, along with `num_moves`. With that, the ones value of the number is obtained by taking the difference between `num_moves` and `tens_value`. Just like that, specific segments of the virtual 7-segment LED are "lit" for any `num_moves`.

12 progress-bar

12.1 Overview

A progress bar is a visually attractive way to display the remaining time for time trial mode. The progress bar module draws a progress bar under the game board whose width reduces at a constant rate along with the remaining time. The game goes to the lost state when the progress bar reaches the left most side.

12.2 Block Diagram

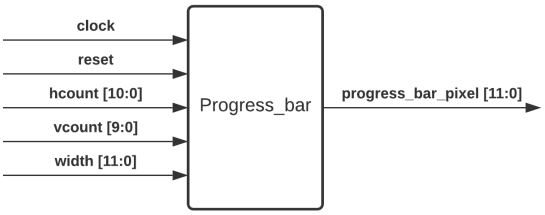

Figure 15: The progress_bar module.

Inputs:

- `clock`: The 65Mhz clock
- `reset`: The reset for the system
- `hcount[10:0]`, `vcount[9:0]`: where we are drawing currently
- `width[11:0]`: dictates the width of the inner rectangle, which is five times the time left in seconds

Outputs:

- `progress_bar_pixel[11:0]`: the 12 bit RGB value that we want to draw at a given `hcount` and `vcount`

12.3 Design

We decided to represent the progress bar as a small yellow rectangle inside of larger blue rectangle. `max_width` is a local parameter that represents the width of the larger blue rectangle, it is also the representation of the

maximum possible time trial duration of 124 seconds. `width` represents how many seconds are left, and therefore decides the width of the inner rectangle. One second is represented by a width of five pixels.

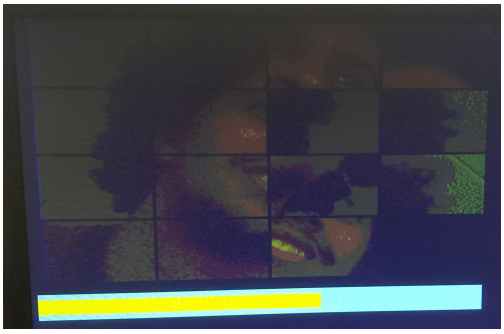

Figure 16: Progress bar.

13 win_blob and lose_blob

13.1 Overview

`win_blob` and `lose_blob` are two similar modules that display an image. Where `win_blob` shows a congratulatory image, `lose_blob` shows a sad face emoji that asks the player if they would like to start again. Depending on the game mode and who won or lost they can be scaled, and moved to different parts of the screen. The image is only scaled if it's a single player mode (classic, time trial, or max moves). For the two player mode, the images can be shown on opposite sides to reflect who won and who lost.

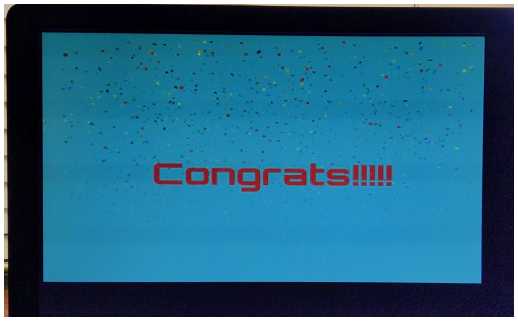

Figure 17: Player won.

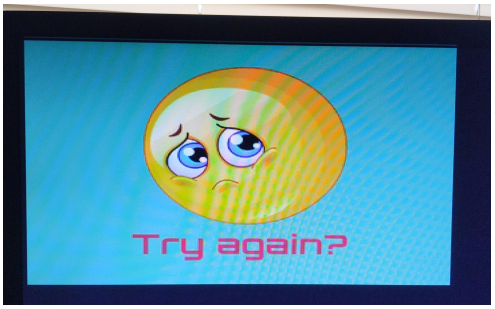

Figure 18: Player lost.

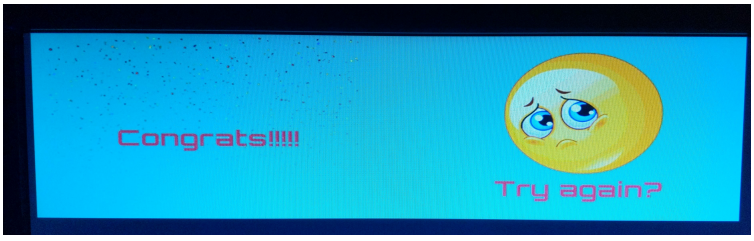

Figure 19: Game ended with player one winning and player two losing.

13.2 Block Diagram

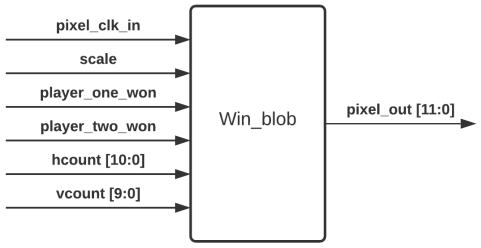

Figure 20: The win_blob module.

Inputs:

- `clk_in`: The 65Mhz clock
- `hcount_in[10:0]`, `vcount_in[9:0]`: where we are drawing currently
- `scale`: decides if we will display 320x240 or 640x480
- `player_one_won`, `player_two_won`: tells us the current status of the ongoing game, an image will only be presented if the two bits are opposites.

Outputs:

- `pixel_out[11:0]`: the 12 bit RGB value that we want to draw at a given `hcount_in` and `vcount_in`

13.3 Design

`scale`, which depends on if the game mode is two player or not, decides if we will be up-sampling the image. If scale is asserted only the status of `player_one_won` decides what is drawn on the screen and only either `win_blob` or `lose_blob` will be drawn because it means we are in a single player mode. But, if scale is not asserted, which tells us that the game is in two player mode, both the winning and losing image will be shown when a game is finished. If `player_one_won` is asserted while `player_two_won` is not asserted, `win_blob` will be drawn on the side of player one (left) while `lose_blob` will be drawn on the side of player two (right), and vice versa. This offsetting is accomplished by modifying the `hcount_in` and `vcount_in` based on an offset.

14 top_level

14.1 Overview

Top level is where it all happens. It is the entry point that allows the game peripherals to interact with the internal states. This module also houses the main modules, the game FSM, and the visual controllers.

14.1.1 Home of the Main Modules

We saw how `grid_shuffle_game` housed many of the modules pertaining to a single instance of the grid shuffle game, however, there are still some components that are not unique to one instance of the game. For one, as mentioned before, in the TWO-PLAYER mode, we pointed out the importance of starting both game boards in the same shuffled state, so the `shuffle` module is one that multiple instances of the game can share. In addition to the two `grid_shuffle_game` instances, other components that live here include the image capturing modules.

14.1.2 Home of the Game FSM

The game FSM also lives here. It controls the flow of the game, and transitions between the various in-game states depending on what screen the user should be seeing. For instance, the system begins in the MENU state, and once the player has locked in their choice of game **size** and **mode**, the system transitions to the CAPTURING state. From there, depending on the selected **mode**, the system goes to one of: CLASSIC_STATE, TWO_PLAYER_STATE, TIME_TRIAL_STATE, and MAX_MOVE_TRIAL_STATE. Likewise, once the game is over the system goes to one of: WON_STATE, LOST_STATE, and TWO_PLAYER_OVER_STATE. Each of these states have unique visuals attached to them.

14.2 Block Diagram

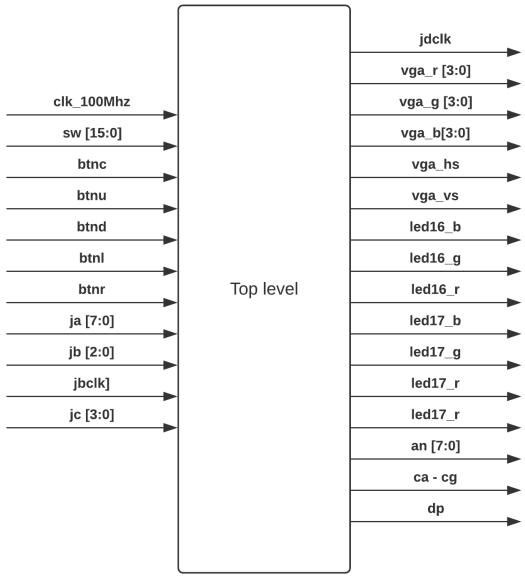

Figure 21: The win_blob module.

Inputs:

- `clk_100mhz`: The system clock
- `sw[15:0]`: The switches on the FPGA
- `btnc,btnu,btnl,btnr,btnd`: The buttons on the FPGA
- `ja[7:0],jb[2:0],jbclk`: Connected to the ESP8266 and used for pixel, clock and other data for the camera

- `jc[3:0]`: Connected to the Joystick inputs

Outputs:

- `vga_r`, `vga_g`, `vga_b`: Used by the FPGA VGA module pixel output
- `vga_hs`, `vga_vs`: represent `hcount` and `vcount` for the FPGA vga module
- `led16_b`, `led16_g`, `led16_r`: LED 16 output
- `led17_b`, `led17_g`, `led17_r`: LED 17 output
- `led[15:0]`: LEDs right above the swicthes
- `ca`, `cb`, `cc`, `cd`, `ce`, `cf`, `cg`, `dp`: cathodes for the seven segment displays.
- `an`: common anode for the seven segment displays.

14.3 Design

Before the it's final design, this module went through several iterations. First, the module was designed solely for the CLASSIC mode. From there, added features, such as TWO-PLAYER mode, introduced complexities that could best be handles using an FSM; that is when the game FSM was introduced.

14.3.1 Game FSM Logic

The game FSM dictates what the player should see at different areas of the game. The FSM begins in the MENU state, where the player can chose a **mode** and **size**. The `mode` selection is controlled by a smaller **menu** FSM that lives inside the MENU state, and the player uses `btnu` and `btnd` to move between the game modes. Pressing `btnu` and `btnd` does not trigger a change of state selection, though; button releases are what should trigger a change of **menu** state. Therefore, on button pressed there is a transition to an intermediary state that is left when the button is released. The same mechanic exists for **size** selection.

Once the player presses button `btnc`, the choices of **mode** and **size** are locked and, similarly, the system transitions to an INTERMEDIARY state, so an effective game state change from the MENU state also happens on button released as opposed to button pressed, before proceeding to the CAPTURING state once `btnc` is released. A `btnc` is used to transition from the CAPTURING state to the **mode** that was selected by the player.

14.3.2 Button Pressed vs Button Released

Now one might wonder why the system transitions on button released instead of button pressed.

- For the game FSM: if we press the center button after it has been debounced, the button will remain active for subsequent clock cycles and the game will trigger unwarranted state transitions.
- For the menu FSM: if any of the directional buttons remain pressed after they have been debounced, the button will remain active for subsequent clock cycles, which would also trigger unintended selection changes, so the player will only be able to select **mode**s and **size**s that are the extremes of their respectful FSM.

14.3.3 Display Logic

The system sends output to the display via VGA; still, the collection of pixels sent depends on the game state. So, in the MENU state, the displayed pixels are those read from the BRAM where the system stores the menu image. In the CAPTURING state, the pixels from the camera frame-buffers stored in a separate BRAM. Finally, in any of the **mode** dependent states, the displayed pixels are those read from the BRAM where the `game_image` is stored. The `top_level` contains most of the modules responsible for displaying visuals because it is easier to multiplex between the different pixel outputs of the modules.

15 Challenges

We initially were worried that this project might not be challenging enough, but as we went through it, we started appreciating the design considerations and decisions it forced us to make. The following are some of the challenges that we faced:

15.1 Timing Challenged

Our initial implementation had issues where our setup and hold times were being violated because we had some combinational blocks that were too big interacting with procedural blocks, so we iterated on our implementation to fix it. We converted most of the combinational blocks we were using to procedural and that forced us to modify some of our implementation for different modules. One design choice that came from this is, some modules have flags that are asserted when they are finished. From this, we got a better understanding of the clock timing report that Vivado provides, and we also got some experience in being able to predict if an issue is timing related or not.

15.2 Running out of BRAM

As we added more and more features, the game had several images for different states, so it was no surprise that towards the end of the project, we ran out of memory. We noticed this when trying add yet another image to serve as game **mode** and **size** selection icons. While this forced us to be a little more creative and to draw those pixels on in real-time, it constrained our ability to integrate audio into the game. It was too late to attempt to use external storage devices. Not having additional BRAM also discouraged us from trying to use the provide COE files for displaying digits 0-9 on the monitor (for the MAX MOVE TRIAL mode), hence why we opted for a little more creative implementation.

16 Possible Improvements / future prospects

If we had more time, we would consider implementing the following features:

- Audio: adding audio clips for different states such as a swoosh sound when making a valid moved, an error sound when a move is invalid, a cheer for winning, and so on.
- Step by step solver: this would be two pronged as we first have to find an optimal solution and then implement the necessary graphics to show the solution. We discussed possible approaches to finding the solution with our industry mentor such as, graph search approaches, constructing a coe file for storing state to sequence of moves mappings, using reinforcement learning and so on
- Other forms of input for players: This semester we were taking 6.810 (Engineering Interactive Technologies) and we considered fabricating inputs such as inkjet printed buttons, a printed menu, or even using what we made for our group project for that class, which was a wearable that detects blinks, and eye movements.

17 Appendix

17.1 Code

The code and necessary resource files for this project can be found at:
`https://github.mit.edu/mussie/grid_shuffle_files`

17.2 Large Block Diagrams

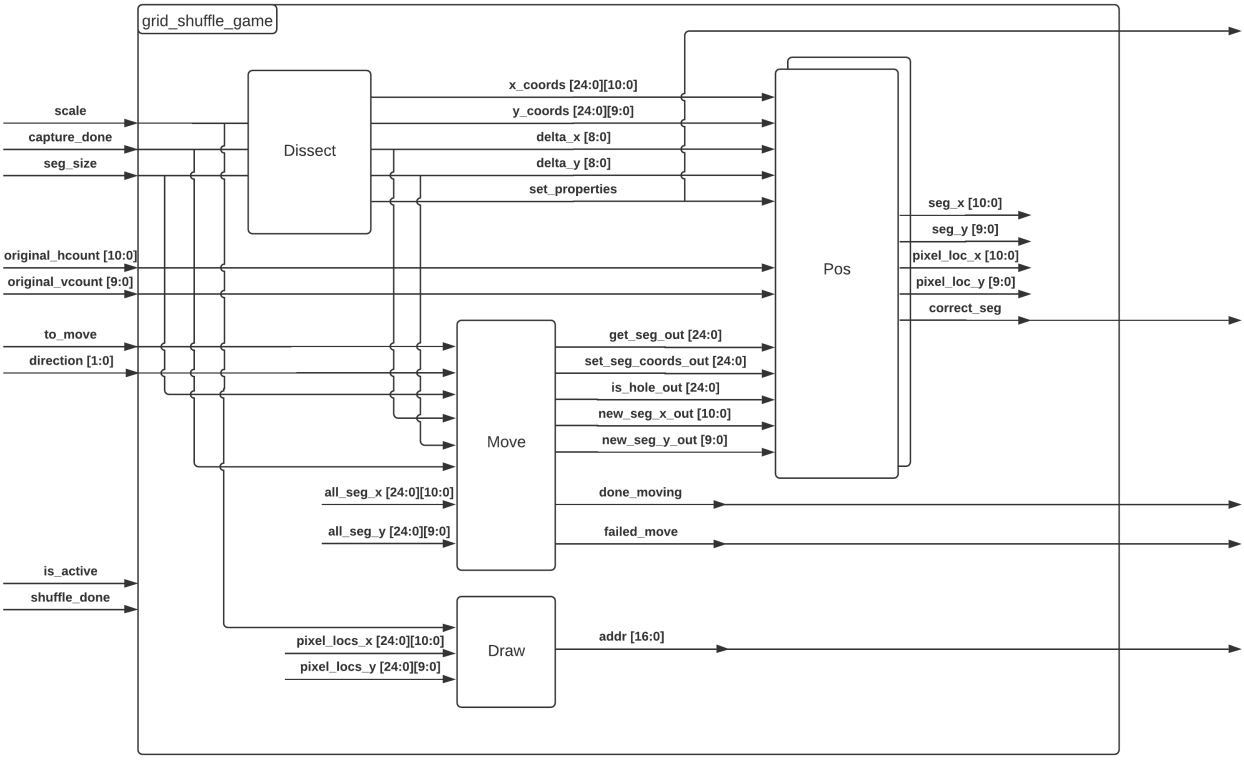

Figure 22: The Internals of grid_shuffle_game

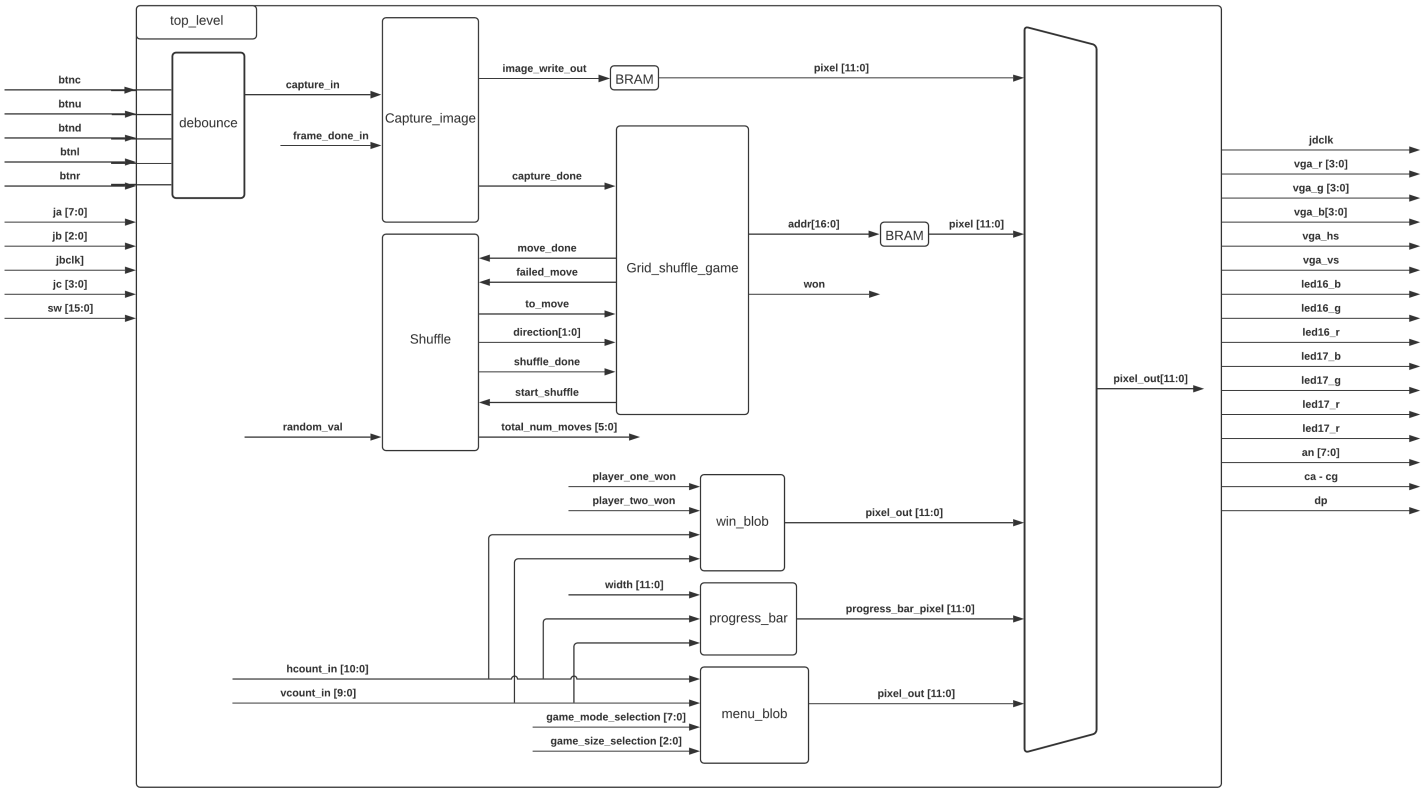

Figure 23: The Internals of top_level