CS M152A Lab 4 Report

Hanna Co, Isha Gonugunta

December 2, 2021

1 Introduction and Requirement

A finite state machine is a computational model that can be used to simulate sequential logic, and represent the control flow. In general, a finite state machine has three components: inputs, outputs, and state transition logic that depends only on the input and the current state.

For this lab, we were given the freedom to design our own project. My lab partner and I chose to re-create mancala, using the Nexys3 Spartan-6 FPGA Board, and use the VGA to display it on a monitor. Mancala is a game that traditionally involves a board made up of two rows of six pockets, each initially holding four marbles. Each player also has a "store" on the two ends of the board, to hold the marbles that they've collected on their turns. The game begins with a player picking up all the marbles from a selected pocket on their side of the board, and moving counterclockwise, depositing a marble in each pocket until they run out. On a player's turn, marbles can be deposited in their own store but their opponent's store is skipped. If the last piece is dropped in your store, you take another turn. If the last marble is deposited in a non-empty pocket, the player picks up all the marbles in that pocket and their turn continues until they drop their last marble in an empty pocket.

To implement this, we displayed the board on a monitor, and pocket selection was done using the buttons on the board. We used four buttons, "enter", "btnl", "btnr", and "rst". The behavior of these buttons are described in the table below.

Button	Behavior
enter	used during the select state to select a pocket
btnr	used during the select state to move right
btnl	used during the select state to move left
rst	restarts the game and resets the board to initial state

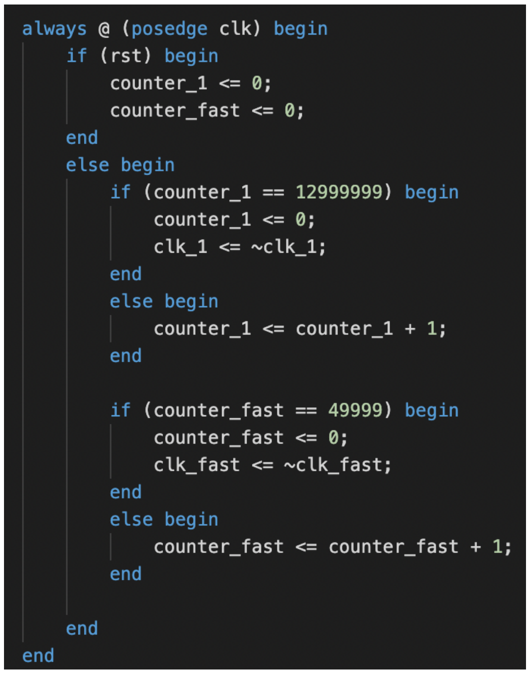
There is one issue when checking for the button press, and that is because of how sensitive the buttons are. Thus, we need to debounce the button press. We do this by creating a three-bit register that stores the past three readings of the button where each is taken on the rising edge of the master clock. We consider the button to have a valid press if we see a rising edge of the buttons value. We check for this rising edge by checking that we see a high reading and the previous reading was low.In addition to the buttons, we also used the seven-segment display on the board to display who's turn it is, along with whether or not we are in a select state.

The monitor would display our board, along with all the marbles. The marble count would be reflected on screen, up to 16 marbles for the pockets, and 48 marbles for the stores. Initially we display 4 marbles in each pocket and empty stores. We also highlighted the current pocket, with the current pocket being what the player is selecting, or the pocket that the player is currently depositing marbles into.

2. Design Requirement

Clocks

Our game required 4 different clocks: one for the seven segment display, one for the VGA board display, one for the marble movement, and finally, the master clock to detect button inputs. These clocks were split into two different modules, `clockdiv` and `diffClocks`. `clockdiv` was taken from the `NERP` example, and `diffClocks` was the clock module we wrote ourselves. Both clocks took in the 100MHz master clock as input, and used counters to output slower clocks at our desired frequencies. The following code snippet illustrates how this is done.



```
always @ (posedge clk) begin
    if (rst) begin
        counter_1 <= 0;
        counter_fast <= 0;
    end
    else begin
        if (counter_1 == 12999999) begin
            counter_1 <= 0;
            clk_1 <= ~clk_1;
        end
        else begin
            counter_1 <= counter_1 + 1;
        end

        if (counter_fast == 49999) begin
            counter_fast <= 0;
            clk_fast <= ~clk_fast;
        end
        else begin
            counter_fast <= counter_fast + 1;
        end

    end
end
```

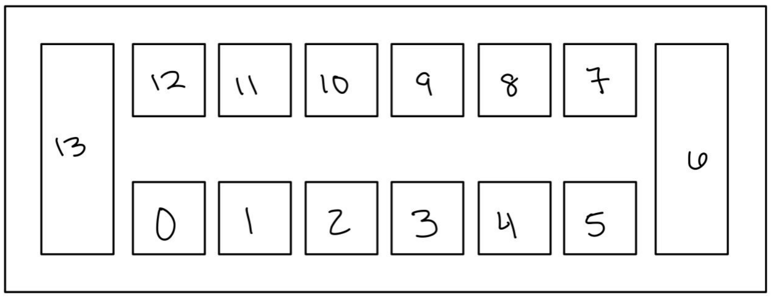
Clock Divider Implementation

As you can see, we increment the counter on each rising edge of the master clock, and invert our clock once the counter reaches a certain value, to simulate a slower clock.

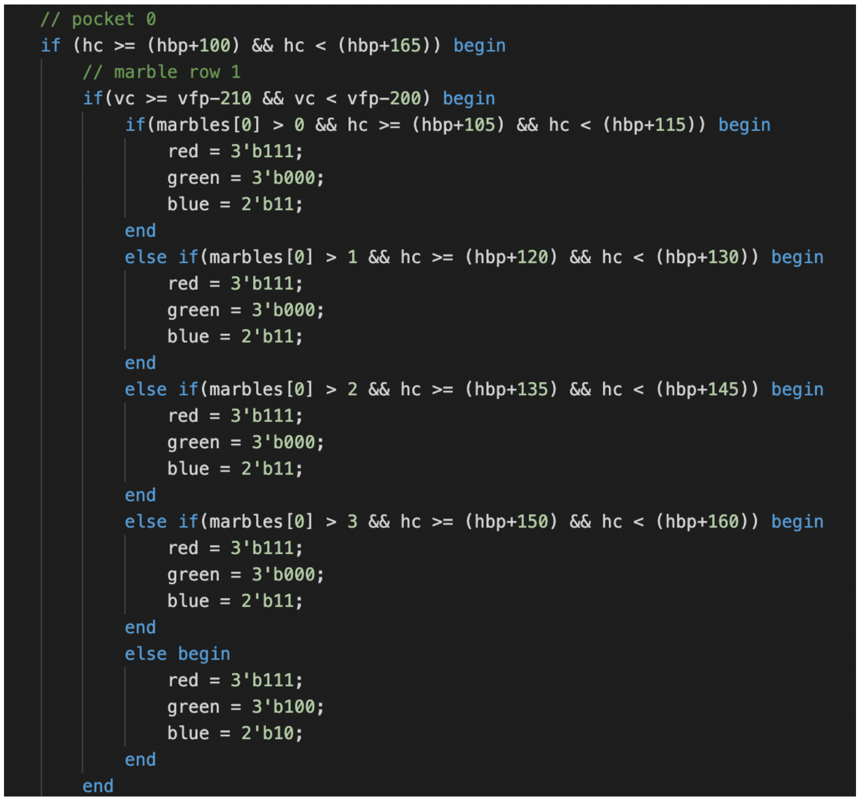
VGA Module

We also had a `vga640x480` module, which was used for our game's on-screen display. We also took this from the `NERP` example, and modified it to our needs. This module takes in a 25MHz clock `dclk`, and asynchronous reset `clr`, the current pocket to highlight `pocket_curr`, and the marble count for each pocket `marbles`. The outputs of this module were not changed. We first modified it by checking the horizontal and vertical constraints of the board, which was ten pixels from the sides, and 130 pixels from the top and bottom of the screen.

We first displayed the stores, which were 20 pixels from the top and bottom of the board and 15 pixels from the side of the board. Each store is 180 pixels tall and 65 pixels wide. Depending on the marble count for each store, we displayed the marbles in rows of four, with each marble being a 10 by 10 square, with 5 pixels between each marble. The pockets were created in a similar way, with each pocket being a 65 by 65 pixel square. The marble display was also determined by the value of `marbles[i]`, with `i` being the index of the current pocket or store. We indexed the pockets and stores in a counterclockwise manner, starting with pocket 0 in the bottom left of the screen. Keeping the marble dimensions the same, we were only able to display up to 16 marbles in the pockets. The pocket numbering is shown in the diagram below.



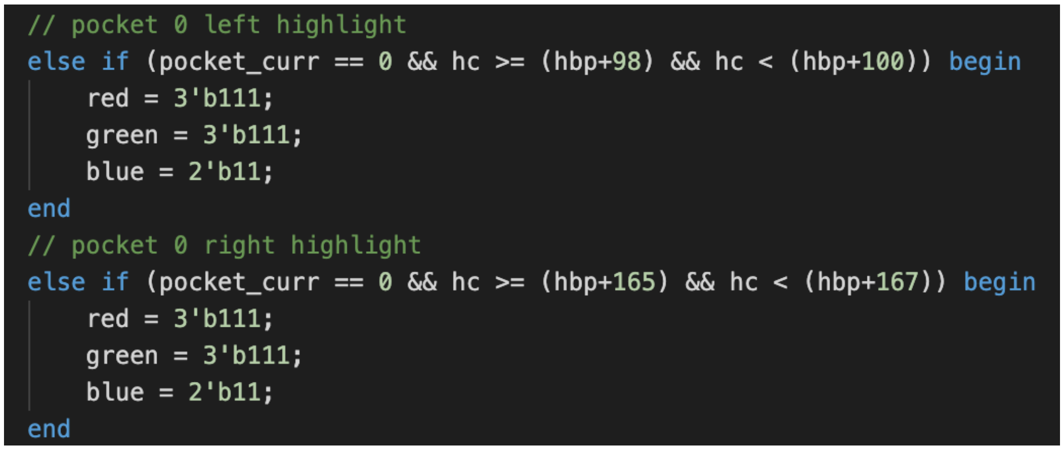
The diagram is from player 1's perspective-- that is, pocket 6 is player 1's store, and `sum1` would hold the sum of marbles in pockets 0 to 5. The code snippet below shows how we decide whether to display each marble based on the current marble count in that pocket. In the case that we have no marbles in a pocket, we simply fall through to the else case, where we fill the square with the pocket color.



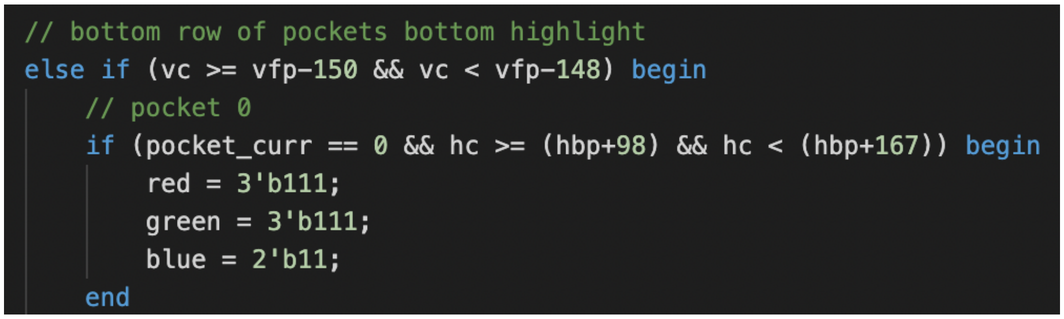
```
// pocket 0
if (hc >= (hbp+100) && hc < (hbp+165)) begin
    // marble row 1
    if(vc >= vfp-210 && vc < vfp-200) begin
        if(marbles[0] > 0 && hc >= (hbp+105) && hc < (hbp+115)) begin
            red = 3'b111;
            green = 3'b000;
            blue = 2'b11;
        end
        else if(marbles[0] > 1 && hc >= (hbp+120) && hc < (hbp+130)) begin
            red = 3'b111;
            green = 3'b000;
            blue = 2'b11;
        end
        else if(marbles[0] > 2 && hc >= (hbp+135) && hc < (hbp+145)) begin
            red = 3'b111;
            green = 3'b000;
            blue = 2'b11;
        end
        else if(marbles[0] > 3 && hc >= (hbp+150) && hc < (hbp+160)) begin
            red = 3'b111;
            green = 3'b000;
            blue = 2'b11;
        end
        else begin
            red = 3'b111;
            green = 3'b100;
            blue = 2'b10;
        end
    end
```

Pocket 0 display for marble row 1

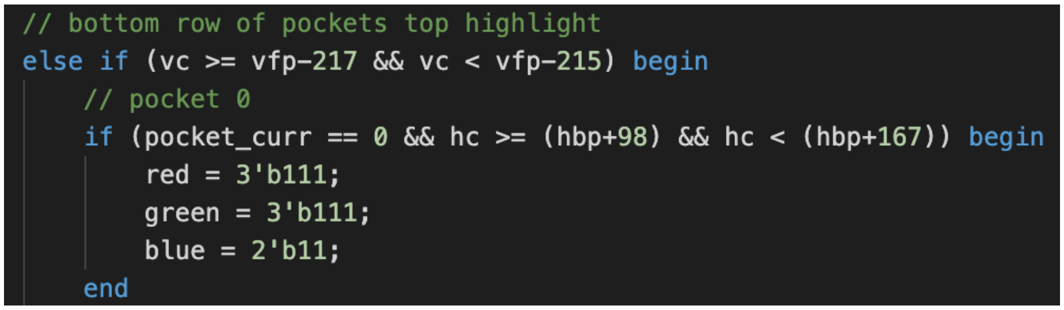
Finally, we took in the active pocket, `pocket_curr` as input, and highlighted the proper pocket based on `pocket_curr`'s value. The code snippet below shows how we check `pocket_curr`'s value and activate a two pixel white border around the active pocket.



```
// pocket 0 left highlight
else if (pocket_curr == 0 && hc >= (hbp+98) && hc < (hbp+100)) begin
    red = 3'b111;
    green = 3'b111;
    blue = 2'b11;
end
// pocket 0 right highlight
else if (pocket_curr == 0 && hc >= (hbp+165) && hc < (hbp+167)) begin
    red = 3'b111;
    green = 3'b111;
    blue = 2'b11;
end
```



```
// bottom row of pockets bottom highlight
else if (vc >= vfp-150 && vc < vfp-148) begin
    // pocket 0
    if (pocket_curr == 0 && hc >= (hbp+98) && hc < (hbp+167)) begin
        red = 3'b111;
        green = 3'b111;
        blue = 2'b11;
    end
```

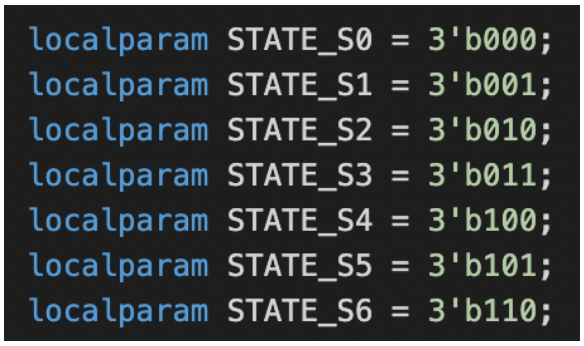


```
// bottom row of pockets top highlight
else if (vc >= vfp-217 && vc < vfp-215) begin
    // pocket 0
    if (pocket_curr == 0 && hc >= (hbp+98) && hc < (hbp+167)) begin
        red = 3'b111;
        green = 3'b111;
        blue = 2'b11;
    end
```

Pocket 0 display for selection highlight

VGA Module

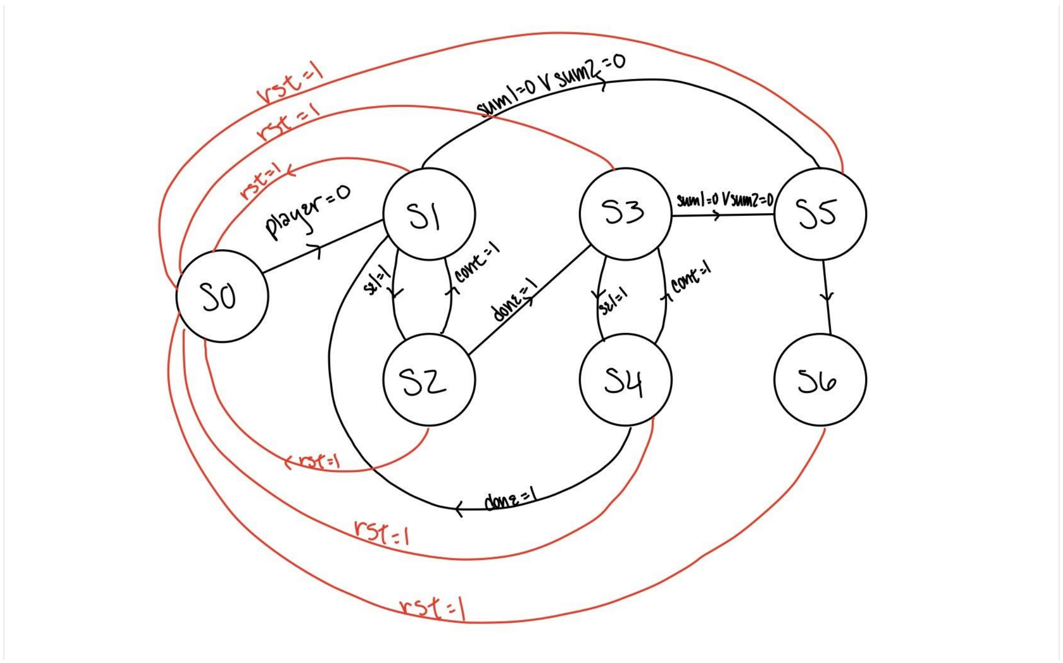
The final module we created was `mancala`, which would dictate the game's functionality. This module took in `clk`, `btnl`, `btnr`, `enter`, `dclk` and `rst` as inputs, and output values for the three anodes we used, the seven segment display, `hsync`, `vsync`, `red`, `green`, `blue`, and `done`. We also defined our seven states as shown below.



```
localparam STATE_S0 = 3'b000;
localparam STATE_S1 = 3'b001;
localparam STATE_S2 = 3'b010;
localparam STATE_S3 = 3'b011;
localparam STATE_S4 = 3'b100;
localparam STATE_S5 = 3'b101;
localparam STATE_S6 = 3'b110;
```

State Declaration

`STATE_S0` is our initial state, and `STATE_S1` and `STATE_S3` are the select states for player 1 and player 2 respectively. `STATE_S2` and `STATE_S4` are the game play states for player 1 and player 2 respectively as well. `STATE_S5` is the state where one player's side is empty, and we move all the remaining marbles to the other player's stores. `STATE_S6` is our final state where we determine and display the winner by comparing the final number of marbles in each store. Our state transitions are shown in the diagram below.

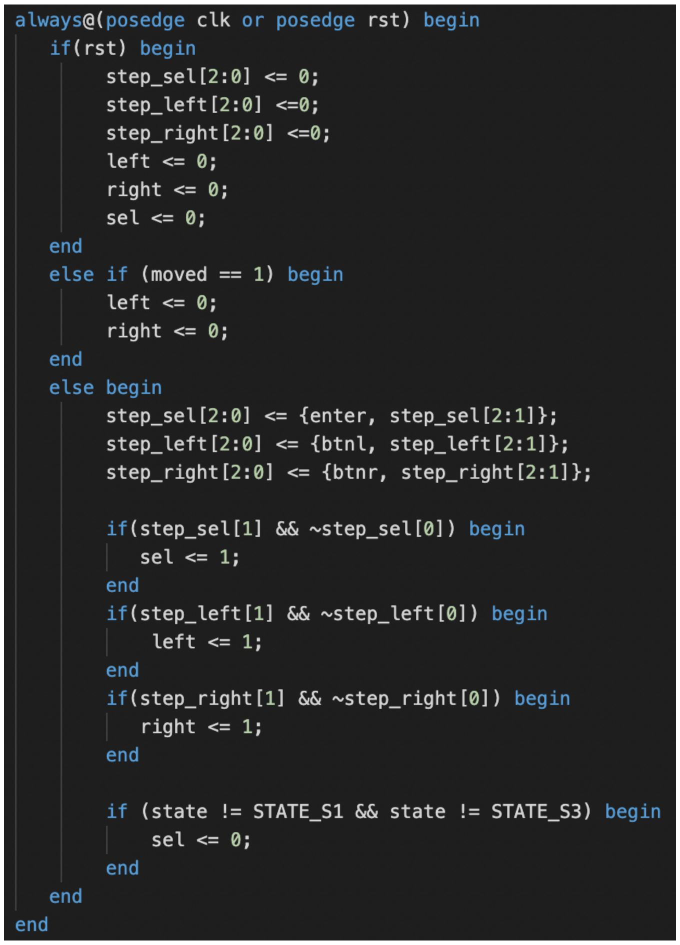


FSM Diagram for mancala game

`STATE_S0` automatically transitions to `STATE_S1`, since `STATE_S0` is a reset state, and we always begin with player 1's turn. We return to `STATE_S0` whenever rst is pressed.

If we're in `STATE_S1` or `STATE_S3`, we check the values of `sum1` and `sum2`, which hold the total number of marbles on each side of the board. If either are equal to 0, meaning that one player's side is completely empty, we transition to `STATE_S5` since there are no more moves for one of the players. In these selection states, the current player is able to use `btnl` and `btnr` on the FPGA to select a pocket. Each player is only allowed to choose pockets in their own row and cannot select their store. Once the player is on the pocket that they would like to select, they can press `enter` to officially select it and make their move.

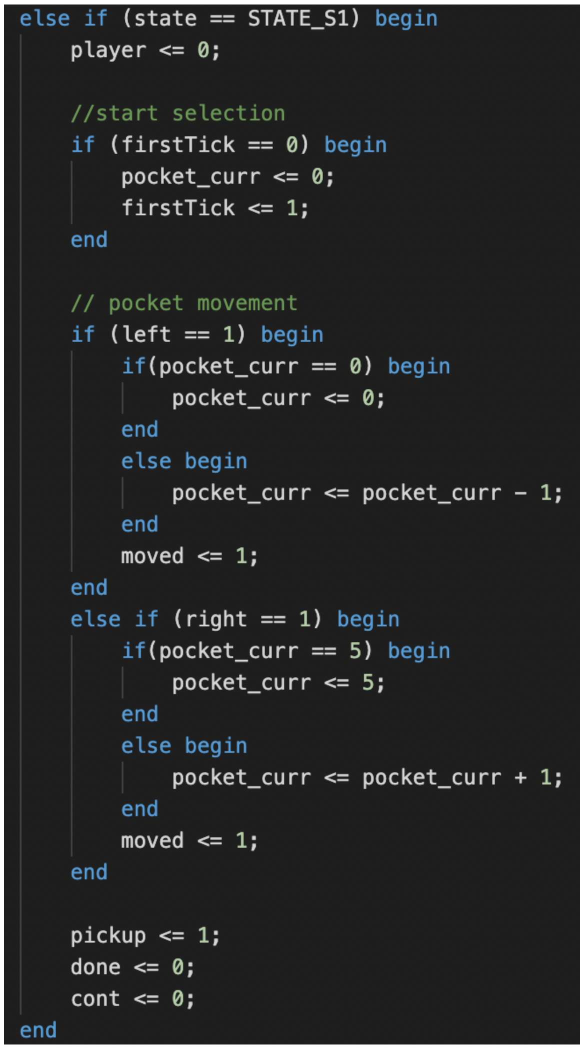
On each rising edge of the master clock `clk`, we check for inputs from our buttons. If we're in one of the select states `STATE_S1` or `STATE_S3`, then we set `left` or `right` to 1 based on inputs from `btnl` and `btnr`. `btnl` moves the active pocket left and `btnr` moves the active pocket right.



```
always@(posedge clk or posedge rst) begin
   if(rst) begin
        step_sel[2:0] <= 0;
        step_left[2:0] <=0;
        step_right[2:0] <=0;
        left <= 0;
        right <= 0;
        sel <= 0;
   end
   else if (moved == 1) begin
        left <= 0;
        right <= 0;
   end
   else begin
        step_sel[2:0] <= {enter, step_sel[2:1]};
        step_left[2:0] <= {btnl, step_left[2:1]};
        step_right[2:0] <= {btnr, step_right[2:1]};

        if(step_sel[1] && ~step_sel[0]) begin
           sel <= 1;
        end
        if(step_left[1] && ~step_left[0]) begin
            left <= 1;
        end
        if(step_right[1] && ~step_right[0]) begin
           right <= 1;
        end

        if (state != STATE_S1 && state != STATE_S3) begin
            sel <= 0;
        end
   end
end
```

Button detection and debouncing

If we are in one of the select states, we set `moved` to 1 if either `left` or `right` is set and change `pocket_curr` in the appropriate direction. We also set `pickup` to 1, which indicates that we need to pick up the marbles whenever we transition to one of the player states. If we press `enter`, we set our variable `sel` equal to 1, to indicate that we should move to one of the game play states.

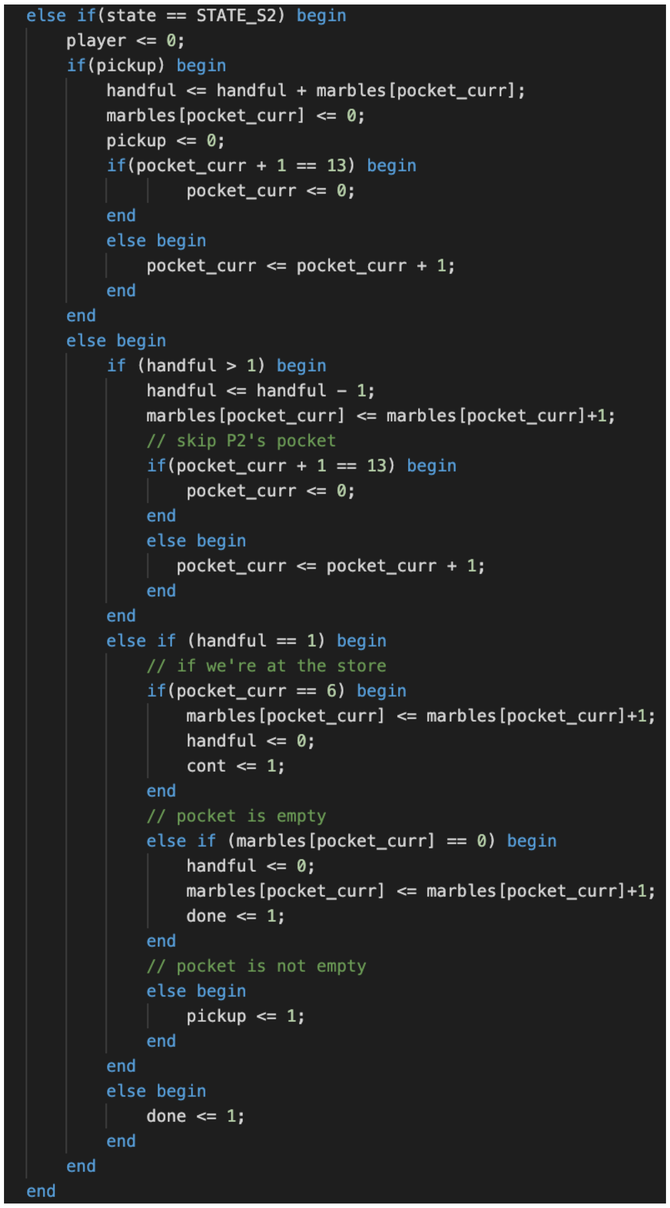


```
else if (state == STATE_S1) begin
    player <= 0;

    //start selection
    if (firstTick == 0) begin
        pocket_curr <= 0;
        firstTick <= 1;
    end

    // pocket movement
    if (left == 1) begin
        if(pocket_curr == 0) begin
            pocket_curr <= 0;
        end
        else begin
            pocket_curr <= pocket_curr - 1;
        end
        moved <= 1;
    end
    else if (right == 1) begin
        if(pocket_curr == 5) begin
            pocket_curr <= 5;
        end
        else begin
            pocket_curr <= pocket_curr + 1;
        end
        moved <= 1;
    end

    pickup <= 1;
    done <= 0;
    cont <= 0;
end
```

Selection state action

If `sel` equals one, we transition from the selection state to the respective game play state, either `STATE_S2` or `STATE_S4`. Once we are out of the selection states, sel is set back to 0. If we are in the game play states, we check the value of `done` to see if we are finished with our turn. If our turn is over, we transition to the opponent's select state. We also check the value of `cont` to see if we need to select another pocket; in this case, we move back to the current player's select state (`STATE_S1 or STATE_S3`) to allow them to begin a new turn. Otherwise, we remain in the current state.

If we are in one of the game play states, we first check the value of `pickup` and increment `handful` accordingly. We move counterclockwise around the board, depositing marbles in each pocket, and skipping the opponent's store. If we only have one marble left, we check to see where we are depositing this marble. If we deposit it in a store, we set `cont` to 1 to indicate that the current player gets to choose another store. If we deposit in an empty pocket, we set `done` to one to indicate that this player's turn has ended. Otherwise, the pocket we deposit in is not empty, so set `pickup` to 1 to indicate that we pick up the marbles in that pocket. The code snippet below is for `STATE_S2`, player 1's game play mode.



```
else if(state == STATE_S2) begin
    player <= 0;
    if(pickup) begin
        handful <= handful + marbles[pocket_curr];
        marbles[pocket_curr] <= 0;
        pickup <= 0;
        if(pocket_curr + 1 == 13) begin
                pocket_curr <= 0;
        end
        else begin
            pocket_curr <= pocket_curr + 1;
        end
    end
    else begin
        if (handful > 1) begin
            handful <= handful - 1;
            marbles[pocket_curr] <= marbles[pocket_curr]+1;
            // skip P2's pocket
            if(pocket_curr + 1 == 13) begin
                pocket_curr <= 0;
            end
            else begin
               pocket_curr <= pocket_curr + 1;
            end
        end
        else if (handful == 1) begin
            // if we're at the store
            if(pocket_curr == 6) begin
                marbles[pocket_curr] <= marbles[pocket_curr]+1;
                handful <= 0;
                cont <= 1;
            end
            // pocket is empty
            else if (marbles[pocket_curr] == 0) begin
                handful <= 0;
                marbles[pocket_curr] <= marbles[pocket_curr]+1;
                done <= 1;
            end
            // pocket is not empty
            else begin
                pickup <= 1;
            end
        end
        else begin
            done <= 1;
        end
    end
end
```

Player state action

If we are in `STATE_S5` and one side still has marbles, we move these marbles into the respective player's store and clear all pockets. After this first cycle, we always move to `STATE_S6`.

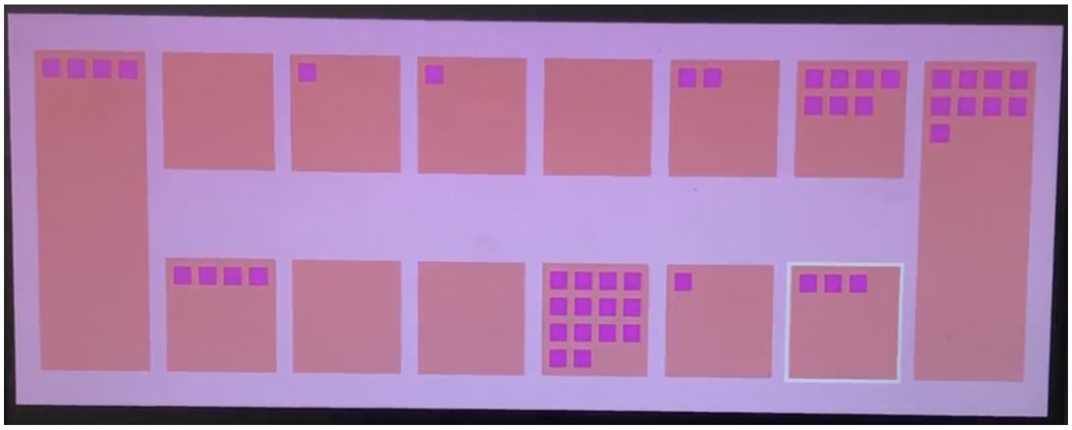
In `STATE_S6`, we determine who the winner is based on the current values in each store. If there is a majority, we display the winning player on the seven segment display. If there is a tie, we simply display P- to indicate that neither player won. `STATE_S6` is our final state, so we stay there until the game is manually reset.

The logic is the same for `STATE_S4`, except we skip player 1's store. If we are in `STATE_S5`, we check the values of `sum1` and `sum2` to see who takes the remaining marbles. Finally, in `STATE_S6`, we set the winner.

Seven Segment Display
The final part of the module is determining what to display on the seven-segment display. We use `digitSelect` to select which anode we are activating on each tick of `clk_fast`. Depending on the values of `digitSelect`, `player`, and the state, we pass certain values to each of the seven segments to display certain letters and numbers. We display a P and the number of the current player in the lower two digits. If we are in `STATE_S1 or STATE_S3` we display an `S` in the top digit to indicate that we are in a selection state. If we are in `STATE_S6` we display the winning player or P- if we have a tie.

3 Simulation Documentation

Since the game was implemented only using the board and VGA display, we checked the functionality of the program by playing a couple of games on the board. We did this to test the button inputs, and to ensure that the pocket highlighting and marble display was correct. Playing through the game also allowed us to check if our states were transitioning correctly, from select to game play, and between each player's turn. We also verified that the seven segment display matched the player's turn, the state, and the winner of the game. The screenshot below is what our VGA looks like.



The marbles are the deeper pink, and the active pocket has a thin white border. The board is a light pink, and the pockets and stores are the salmon colored areas.

We tested for several behaviors. We ensured that player one could only move between pockets 0 through 5 and player two could only select from pockets 7 to 12. We also checked that the pocket selection couldn't move out of bounds. For example, if the current pocket was pocket 5, pressing the right button would not change the pocket. We also checked that the highlight started at the player's leftmost pocket at the start of their selection state. We also checked that landing in the store would give the player another turn.

We also checked for specific edge cases. One case was the selection of an empty pocket. We chose to make it so that a player's turn would end if they selected an empty pocket.

4 Conclusion

To summarize, we used two clock modules, one that was part of the `NERP` example and one that we created. They both took in one master clock, and used that to create the other clocks we needed for our game. We also modified the `NERP` module to display our board and all the marbles on the monitor. Our main module, `mancala`, implemented all the logic for game play, including selecting pockets, marble movement, changing turns, and the end of the game.

We ran into several problems when implementing the `mancala` module. For one, when a player's turn ended, the states weren't transitioning correctly. However, we realized that this was because we used multiple `if` statements, instead of using `if else if` statements. Another problem we encountered was that our seven segment display wasn't displaying in the correct order. Instead of displaying "`S  P 1`", it would display "`P  S 1`". This was because we initially had a module for displaying on the seven segments, however, we got rid of this, as it was an overcomplication. We simply integrated it into our code, which solved our issue. Another issue we faced at first was coordinating reading in the button inputs and highlighting the current pocket. Initially, we were modifying the pocket whenever we saw a valid button press, but this wasn't reflected on the display until the next tick of clk_1. Since we checked the value of the button on the faster master clk, our movement appeared to skip some pockets because it was hard to tell whether a button press had been registered or not before pressing again. Changing it to only allow for one movement per clock cycle fixed this issue.

The most challenging part of this lab was figuring out how to integrate the `vga640x480` and `mancala` modules, and how to find ways to make the states transition properly. Since we had creative freedom for this lab, I don't have any suggestions relating directly to the lab. However, one suggestion I will make is to have the report due finals week, as many groups don't finish until the second to last lab section, and it's difficult to begin writing a report without finishing the project. By making it due sometime during finals week, this allows groups to finish their projects.