Lab 4 Report

Aparna, Nicole, Paul

Introduction

In this lab, we created multiple modules to successfully implement a functioning game of “Whack-a-mole” on an FPGA board. We used the 7-segment display in order to display the target number and our stopwatch. Additionally, we used an accessory pmod SSd attachment that displayed the points accumulated by the player. This project was implemented using Verilog HDL.

Our version of “Whack-a-mole” uses a random number generator that creates our target number. Alongside our target number is a stopwatch that displays numbers 0-99 at a fast speed. When the stopwatch number is close or equal to our target number, the player should press the button to pause the stopwatch and “whack the mole”. Depending on how close the paused number is to the target number, the player is awarded points.

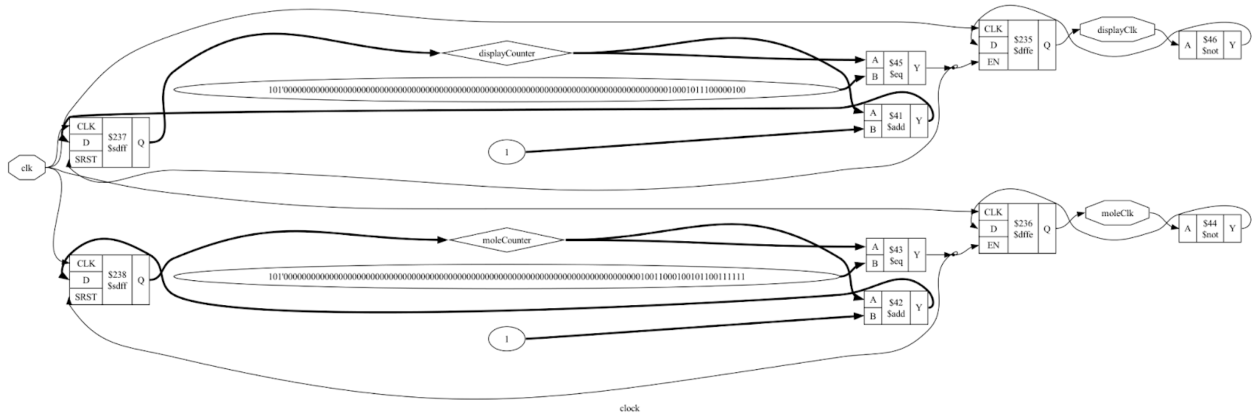
In this lab, we created seven modules: clock, counter, debounce, display, random, pmodSsd, and main. The clock module kept track of the different speeds of the random target number being generated and the stopwatch display. The counter module kept track of how the stopwatch was incrementing as it was being displayed. The debounce module focused on allowing the stopwatch value on the board to be saved with the press of the button. This allows players to pause the stopwatch without holding it down. The display and the pmodSsd module configure the different bits of the 7-segment display and the pmod SSd accessory attachment to display the target number, stopwatch number, and score. The random module creates the random number generator that is used to display our target number. Lastly, the main module configures the switches, buttons, and other portions of the display that bring the game to completion.

Simulation

No simulation was used during this lab. All testing was done directly on the Nexys 3 FPGA board. We tested all the separate components of the lab. First, we made sure the randomizer that generated the “mole” worked correctly and returned a different two-digit number each time. We then tested the counter by adjusting the clock until we could see the numbers changing clearly. We tested debouncing by hitting the “whack” button and making sure it stopped the counter. Lastly, we made sure the pmod ssd devices worked currently by calculating what the score should be each time we “whacked” the mole, then comparing it to what was displayed on the left pmod ssd. We also tested that changing the player switch changed the number that was displayed on the right pmod ssd and reset the score.

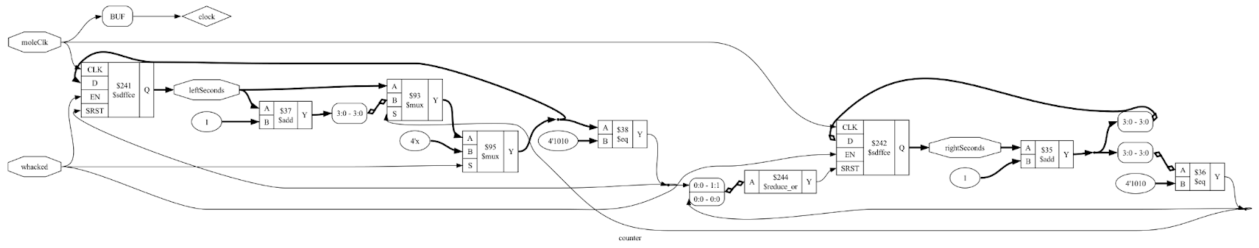
Design

clock



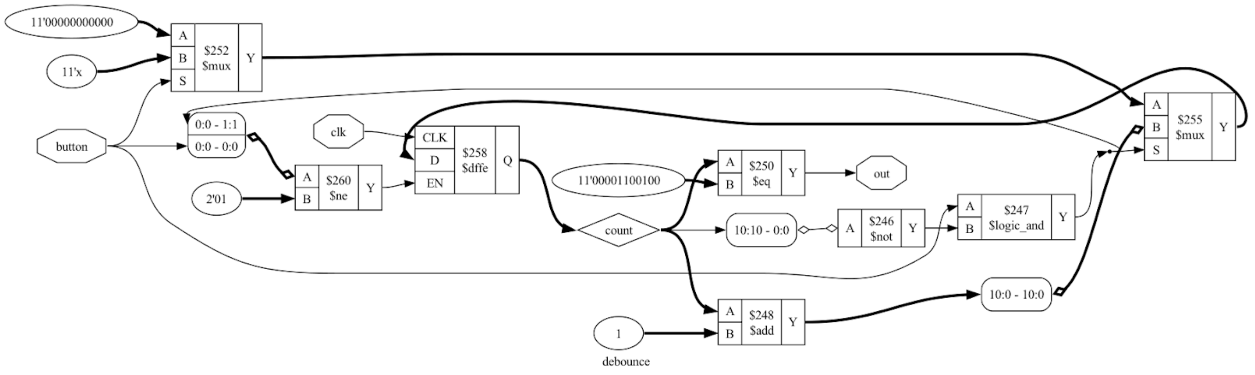
The clock module takes in a register input clk and outputs a register displayClk and register moleClk. The input clock is the 10 MHz clock that is the standard clock that accompanies the FPGA board. Within the module, we created temporary registers displayCounter and moleCounter. These counters were initialized to 0 and incremented by 1 at every positive edge of the clock. When the moleCounter hits 10 Hz, we set it back to 0 and flip the moleClk's value. This way, the moleClk will blink at a 10 Hz speed when randomizing the target number. Similarly, when the displayCounter hits 700 Hz, we set it back to 0 and flip displayClk's value. In this case, displayClk is incrementing the faster moving stopwatch clock.

counter



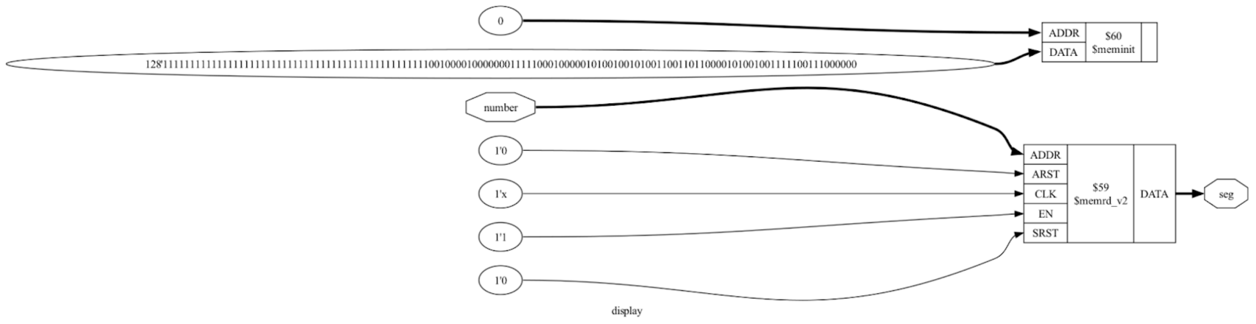
The counter module takes in the moleClk as an input along with the signal of whether the player pressed the button or not. This module outputs the registers that display the stopwatch number, rightSeconds and leftSeconds. Whenever the moleClk was 1, if the button was not pressed, the counter would keep incrementing. Firstly, the rightSeconds value would increment until it hit 10, and once it did, leftSeconds would begin incrementing. This would keep happening until the stopwatch number reaches 99, at which point the stopwatch returns back to 0.

debounce



The debounce module takes in an input 10 MHz clk and the button that the player must press to pause the stopwatch. The module outputs the value of how long the button must be held for to debounce the button. Through testing different values, we felt as though 100 ns was the best amount of time to hold the button in order to pause it. We created a 10 bit register called count in order to keep track of how long the button was being pressed. At every positive edge of clk, we check if the button is pressed and if the entire 10 bit register is not full. If so, the count register keeps incrementing. This is how we keep track of how long the button should be pressed for in order to register the value at a steady state. If the button is pressed again to unpause the stopwatch, count is set back to 0.

display



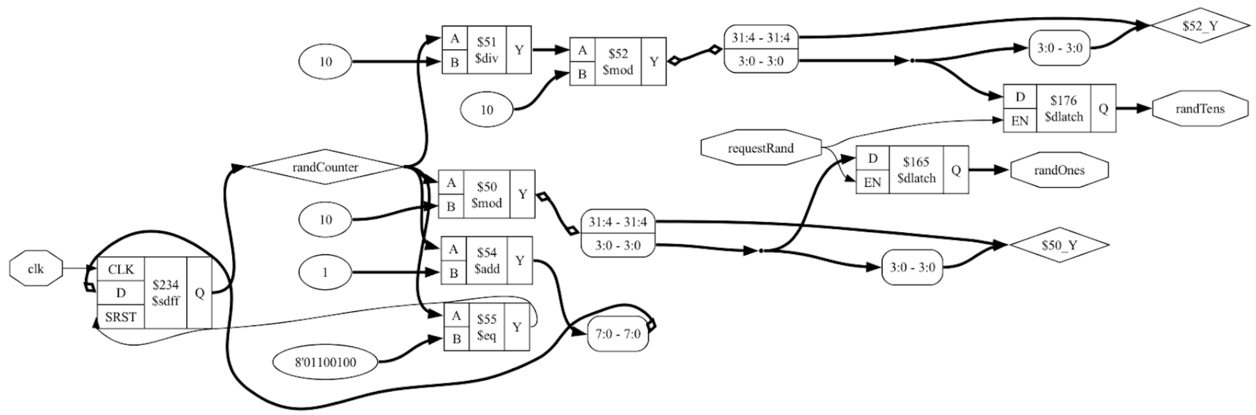
The purpose of the display module is to allow our values to appear on the seven-segment display on the fpga board. We display the target value, which represents the mole, on the left, and the fast counter on the right. We want to “whack” the mole when the counter is close to the target value.

The display module takes a 4-digit number as input and outputs an 8-bit variable called seg, which is then passed to main to actually display the number. Depending on the input number, we have a case statement that matches numbers 0-9 to a particular seg output:

```
always @(*) begin
  case(number)
    4'd0: seg = 8'b11000000;
    4'd1: seg = 8'b11111001;
    4'd2: seg = 8'b10100100;
    4'd3: seg = 8'b10110000;
    4'd4: seg = 8'b10011001;
    4'd5: seg = 8'b10010010;
    4'd6: seg = 8'b10000010;
    4'd7: seg = 8'b11111000;
    4'd8: seg = 8'b10000000;
    4'd9: seg = 8'b10010000;
    default: seg = 8'b11111111;
  endcase
end
```

We call display for tensMole, onesMole, leftSeconds, and rightSeconds so we can display each of these values on the fpga board.

random



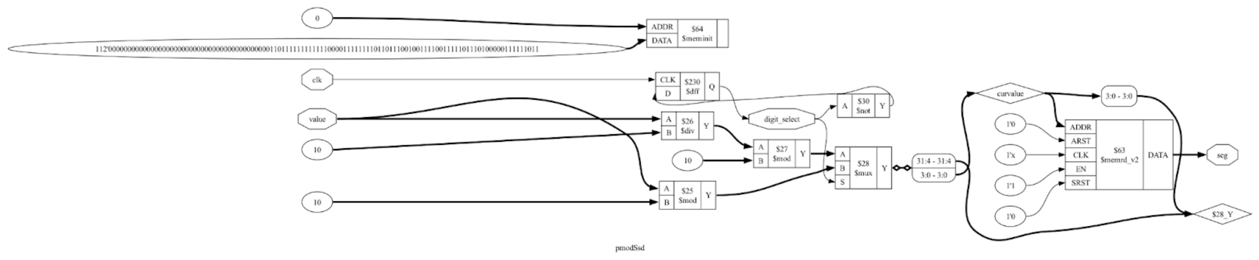
random

This module allows us to display a random target value on the left side of the seven-segment display on the fpga board. Here are the inputs and outputs:

```
module random(
  input clk,
  input requestRand,
  output reg [3:0] randOnes,
  output reg [3:0] randTens
);
```

We pass in a very fast clock to this module. We have a counter that increments whenever the fast clock changes. If requestRand is 1, we set the randOnes and randTens output based on the counter. Since the counter changes extremely fast, these values are effectively randomized. The requestRand input is dependent on a switch on the board, so whenever we flip the switch, we should get a random value. The randOnes and randTens variables are passed into the display module so they can be displayed on the FPGA to represent the mole.

pmodSsd



The pmodSsd module is similar to the display module, but it was quite difficult to figure out exactly how it worked and how to display our values correctly. We first needed to update our UCF file; we uncommented JA[3:0], JB[2:0], JC[3:0], and JD[2:0], because this is where the pmod SSD devices connected to our board.

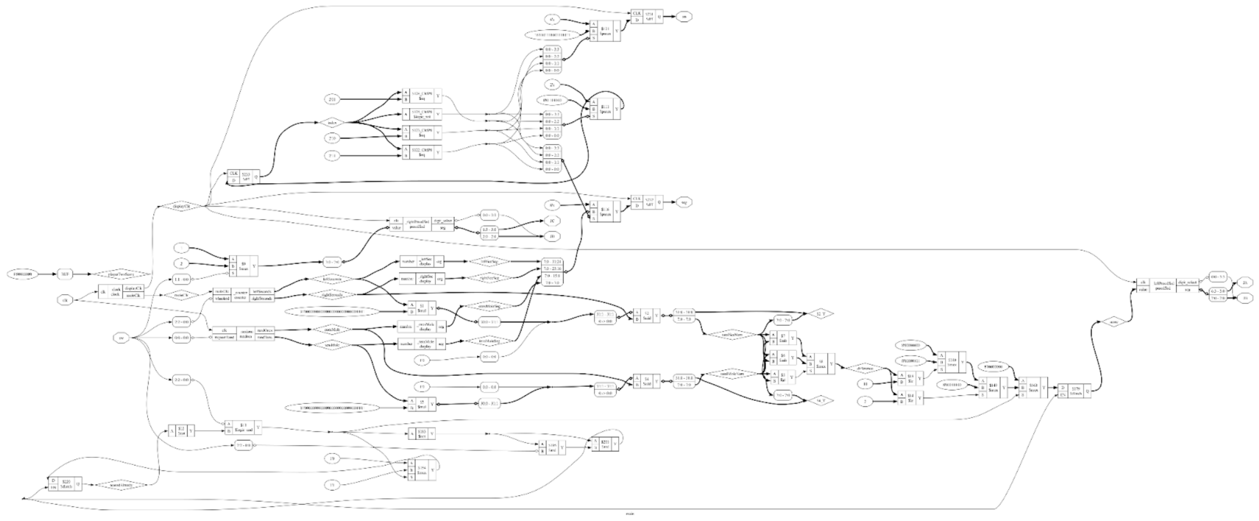
The module takes in the displayClock and the 8-bit value to be displayed as inputs, and outputs variables “seg” and “digit_select”. Similar to the display module, we have a case statement to map each value 0-9 to a particular bit pattern for “seg”. We manually figured out which bits to set to 0 and which bits to set to 1 depending on which segments should be lit up. Digit_select determines which digit in the pmod SSD display we are currently working with, and it alternates based on the displayClock.

We use two pmod SSD display devices and call them leftPmodSsd and rightPmodSsd. Here is how we called the module in main:

```
pmodSsd _leftPmodSsd(
  .clk(displayClk),
  .value(score),
  .seg({JA[3:0], JB[2:0]}),
  .digit_select(JB[3])
);
pmodSsd _rightPmodSsd(
  .clk(displayClk),
  .value(sw[1] ? 2 : 1),
  .seg({JC[3:0], JD[2:0]}),
  .digit_select(JD[3])
);
```

Since we passed the score variable into the leftPmodSsd, the left device displays the score. For rightPmodSsd, we pass in either 2 or 1 as the value, which represents the current player. So the right device displays which player is currently playing. Lastly, the seg outputs are stored in JA, JB, JC, and JD, which allows the values to appear on the device (recall that we updated the UCF file earlier).

main



main.v is where the top-level module, where the magic happens. The high-level logic for the whack-a-mole game is implemented here. Every other module is called from main at least once.

random is called to provide random values, clock is called to provide clock variables of varying frequency, display is called for the built-in seven-segment display, and pmodSsd is called for the external Pmod SSD display.

Inputs are the Nexys 3's 100 MHz clock and an 8-bit array representing the slide switches. Outputs are wires that control the Nexys 3's seven-segment display and external Digilent Pmod SSD seven-segment displays.

There are two always @ blocks.

The first always @ block is sensitive to the third switch (sw[2]), which when enabled represents "whacking the mole." sw[2] is also passed to the counter module; when enabled it pauses incrementing. Inside the always @ block, the score is set based on the absolute difference between the mole value and the counter value (difference is a wire set using an assign statement). If the difference is less than or equal to 2, the score is set to 20. If the difference is less than or equal to 10, the score is set to 5. Otherwise the score is set to 0. The value of score is displayed in the left Pmod SSD display.

The second always @ block is sensitive to the display clock variable and controls the built-in display. It displays the mole value in the first two digits of the display and the counter value in the second two digits.

Conclusion

Overall, this lab was a great learning experience and allowed us to utilize our creativity and problem-solving skills to construct a finished product. It was exciting to think up new features we could add to the game and work toward implementing those features. Although it was a quite difficult lab, we were able to turn toward online resources and previous labs to write our code.

We ran into several difficulties during this lab. When we were first implementing the random module, we wanted to use a Verilog library called "random". However, we ran into several issues and came to the conclusion that our version of Verilog did not support all features of that library; thus, we had to turn to another solution for that module. Another difficult aspect was understanding how to integrate the pmod SSD seven-segment display. For instance, in our journey to display a correct value on that device, we ended up accidentally displaying hex values instead of decimal values, then displaying the values upside down. We eventually decided to write our own bit patterns for each value based on which segments we wanted to be lit up.

Our most difficult challenge was trying to display the scores for player one and player two. We initially wanted to display both scores on the 2 pmod ssd devices, but for some reason, it would display garbage values that we could not successfully decrypt. It was very frustrating because when we plugged in particular values to the leftPmodSsd and rightPmodSsd, like the player one score and the result of a certain button press, the values would display correctly. But then when we changed the inputs to the player one score and player two score, only one of the pmod ssd devices would display the correct value. We were unsure about why changing the input could mess up other areas of our code. We adapted by displaying the score on one device and the player (1 or 2) on the other.

This was a fun and interesting lab overall, but I think we could have benefited from more time, more support beyond just searching things up, and perhaps more detailed requirements.