Shikha Mody (604922996)

Brendon Ng (304925492)

Armaan Singh (404932331)

COM SCI M152A Lab 5

TA: Logan Kuo

**Lab 4 Report**

**Whack-A-Mole**

**Introduction**

For our Lab 4 project, we implemented a Whack-a-Mole game on the FPGA board. We used the switches, LED lights, seven segment display, buttons, and different Verilog modules to implement this design. The mechanism for this game is as follows (from our project proposal):

Each individual LED represents a “mole.” The switches correspond to the “whack.” Points are earned by quickly flipping the switch corresponding to which LED light is on. By flipping the switch (either from OFF to ON or ON to OFF) within a certain amount of time, the mole is effectively “whacked,” meaning that the LED now turns off and another random LED turns on. The score will be displayed on the seven-segment display of the FPGA board, along with which level has been selected. Each mole that is whacked within the time limit will add to the point total. If a mole gets missed (i.e. not whacked within the time limit), or incorrectly “whacked,” the game is over. Buttons will be enabled for “play again” functionality (essentially reset the game), and choosing different levels of difficulty (easy, medium, hard). Each level will have a different amount of time, or multiple LEDs that can light up simultaneously before the mole expires and the whack is missed, effectively ending the game. We also implemented scrolling on the seven-segment display for clearer instructions on gameplay (i.e. saying “Choose level” at the beginning of each game, or “Fail” if the game ended). In addition, we added “hi score” functionality to save each level’s highest score, and notify the player if they have achieved the highest score so far (all through the seven segment display).

This project combines skills we have learned in previous labs. We had multiple internal clocks, similar to the functionality we implemented during Lab 3, different Verilog modules for playing the game, a random number generator for randomization of the LEDs, and a state machine for the seven segment display as we did in Lab 3. The main components of gameplay represent the most important features of the game. Correctly registering a “whack” of a valid “mole” is the basis of the game, as is indicating a “game over” state. The underlying functionality of timing, level selection, screen display features, and other components (i.e. buttons, randomization, etc.) are also components that must be implemented. All in all, this lab combines aspects that we have learned from all of our other labs into one comprehensive game-playing experience.

**Design Description**

We chose to implement this project in a very modular, broken up fashion because it allows for more compartmentalizing and flexibility while having cleaner code with modular development.

Main Module: whackamole

The main module calls all other modules and saves their inputs/outputs as variables. It is here that the hardware clk is kept and passed in to debounce.v where we receive the slower clocks from blackbox modules. The game functionality runs in an always block triggered on every hardware clk cycle. It is essentially a giant state machine where our game flows through 4 stages: beginning, ready, playing, and game over. Once reset is pressed, the game starts off in "beginning" mode, where the user is prompted to choose a level. When a level is selected by the push of one of the three level buttons, we transition to "ready" mode where the display will show the level selected and the high score for that level. In this mode you can now press the up button signalling "start" to transition the game state to "playing" where the actual game will happen. When the player fails (discussed later on), the game state changes to "game over" where FAIL will be displayed along with the final score and whether or not a high score was achieved. From here, a push of the start button takes us back to playing for a new game, a push of the level buttons takes us back to ready and the reset button takes us to beginning (without resetting the high score).

The actual gameplay happens on every clock cycle if our state is not game over. We have a clock called "slowPulse" that has a frequency of 0.5-1Hz depending on the level selected and slowPulse is high for only one hardware clk cycle at a time. So when this pulse comes, it will align with a hardware clk cycle in the main module and one or two random numbers from 1-8 will be generated using random.v, converted to one hot using our oneHot function and is set as the next "moles". What we have is a one or two hot 8 bit value of the switches that will have the next mole. We set a variable for the current state of the switches and a variable for the desired state of the switches which is just the current state with the moles switched. To represent this we used an XOR of current state and the oneHot representation of moles. A buffer counter is also initialized to zero and will start counting up every clock cycle to time the amount of time taken to "whack" the mole.

On every clock cycle, we check for an incorrect flipped switch. We ignore the switch(es) that are the current moles because they can either be in the previous state or the desired state. If any of the other switches do not match the previous state, we immediately switch to game over and the player has failed.

One bug that we ended up fixing was related to the hardware and software of the switches triggering other gameplay actions. We initially let the mole switches have a value of "previous value" or "desired value" since you are allowed a small time buffer to switch them. However, we were coming across the case where we would flip the desired switch and it would immediately show an incorrect flip and game over. We eventually realized that there may be some split second that a clock cycle could align with where the physical switch on the FPGA board is being switched from one side to another and loses its electrical contact with either side, as it was disconnecting from one side and about to connect to the other. This lack of contact would lead to a high impedance (Z) value that is not equal to the previous or desired state of the switch, giving us a failure. We then decided to not trigger anything based on the switches themselves.

If no wrong switch was flipped, the game checks to see if the current state of the switches is equal to the desired (correct) state. If this is the case, our buffer timer stops counting, the score is incremented and we wait for the next mole. If we are still waiting for the desired state to be reached, we count up on the buffer counter. If the buffer counter exceeds a certain threshold determined by the level, the user has taken too long to whack the mole and game over is also reached here. On game over, all LED's are lit up and the game state is changed.

Randomize module:

On every clock cycle, the randomize module outputs a random number between 1-8 (inclusive) using a psuedo-random technique. It involves an initial number being chopped up bitwise, the bits being moved around and modulo'ed in many places to have the effect of a random number without being truly random as the hardware does not support that. This combined with the random times that we will pull the random numbers generated by this module will have the effect of a random number generator. This waveform can be seen in the simulation section.

Clock Dividing

This module takes in the hardware clk and outputs a slowPulse, and a debounceClk for button debouncing. The debounceClk is created like any other normal divided clock with a counter and a target value. When the target value is reached, debounceClk’s value is flipped and counter goes back to 0. The slowPulse clock is slightly different. Since we want this to come at a much slower rate than the hardware clock but only for one hardware clk cycle, we set slowPulse to be the result of a boolean checking if counter (incrementing at every clk) modulo'ed with a certain number that determines the frequency is equal to zero. When this modulo is reached, we reset counter to 1 (not zero or else we would get counter%constant = 0 again). The result can be observed in the simulation documentation section of this report. The clock dividing module lended itself to button debouncing, too. We used a debounceClk to sample each button at a slower rate (similar to previous labs) to debounce the signals.

Display module

The display was implemented similarly to how it was in lab 3 where we have a variable for each of the four digits which held a number 0-9 or for values 10+, they corresponded with a letter. These values when passed in to a function called "digToLed" will be translated from their digit form to a seven segment 8-bit binary value. The display state machine flips through each digit so fast that the human eye cannot detect it, and flashes each digit one by one, rotating through the four digits to make it appear as if all four are illuminated at once.

The value of each of the four digits is determined by a small state machine depending on game state. If game state is "playing", the four digits are always "L", then the level number, then the two digit score. If game state is in “ready” or “game over”, the digits depend on a ticker message which is a large state machine that depends on the game state and values such as level and score. We formed a ticker for many of the stages which involved a ticker clock at around 2Hz that would show four digits and on each cycle, move the 3 last digits up one spot and have a new digit in the rightmost spot which would be the next letter in the ticker. The ticker would be a long string of states that would eventually wrap around back to the start.



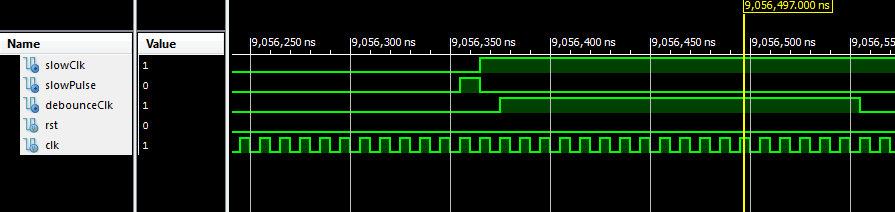
*Figure 1: Display Ticker State Machine*

**Simulation Documentation**

In order to keep our code organized, our lab was broken into a few relatively small modules. Our overall approach to design utilized incremental development. This allowed us to design a feature along with its tests independent of the rest of the project, and then integrate once it is fully functional. First, we designed the switches and lights to show random LED’s. Then we implemented the timing out of a round with a clock divider. And finally we designed our display to choose a level and show a game over when the player fails a round.

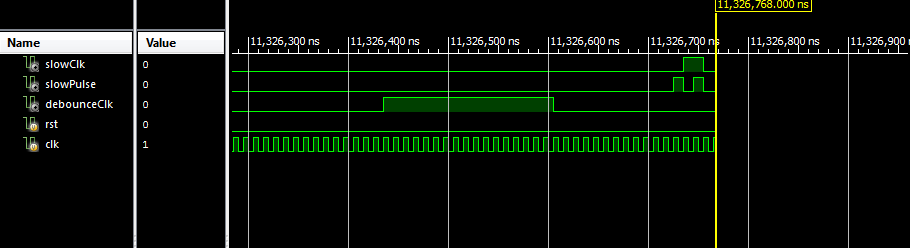
To properly test our project, we simulated each of our modules in the implementation to analyze the produced waveforms, as well as implemented it onto our FPGA board to test it. By testing the individual parts separately we were able to find any bugs we did have more easily and solve them.

The first module we tested was the clock divider, clkDivide.v, using the testbench clkDivide\_tb.v. We aimed to ensure that the various clocks we were using were flipping from 0 to 1 after the correct number of actual clock cycles. In addition to the various clocks, we aimed to debug the slowPulse as well. Below is the waveform of the working clock divider:



*Figure 2: A section of the waveform for the clock divider. slowPulse is only active for one clock cycle, while slowClk and debounceClk are slower clock cycles themselves.*

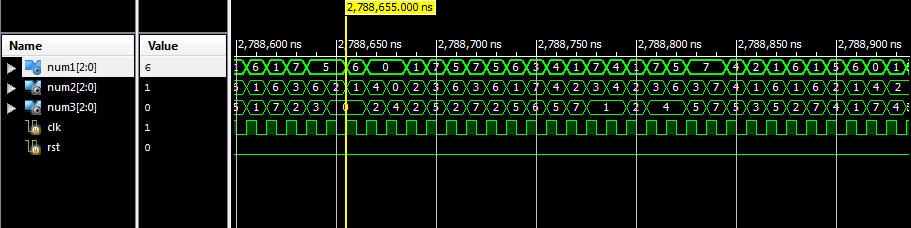
This waveform shows, from top to bottom, the slow clock, slow pulse, debounce clock, reset, and clock of our clock divider module. We can see that the slow clock and debounce clock oscillate much slower than the regular clock. In addition, we can see the slow pulse turn on for a single clock period. Prior to debugging, we had an error which could cause the game to increment the score multiple times during a turn. Upon reviewing the buggy waveform of the clock divider:



*Figure 3: Clock divider waveform with buggy slow pulse*

Here we can see that the slow pulse is set to 1 twice, instead of once. This was because the slow pulse was set to one if a modulo of a counter variable be evaluated to 0. If so, the counter was set back to 0. This was a problem as it ensured that the result of the mod operation would essentially be 0 twice, hence the double 1 appearing in the waveform. By resetting the counter to 1 instead, we ensured that the slow pulse would function as intended, adding consistency to the score.

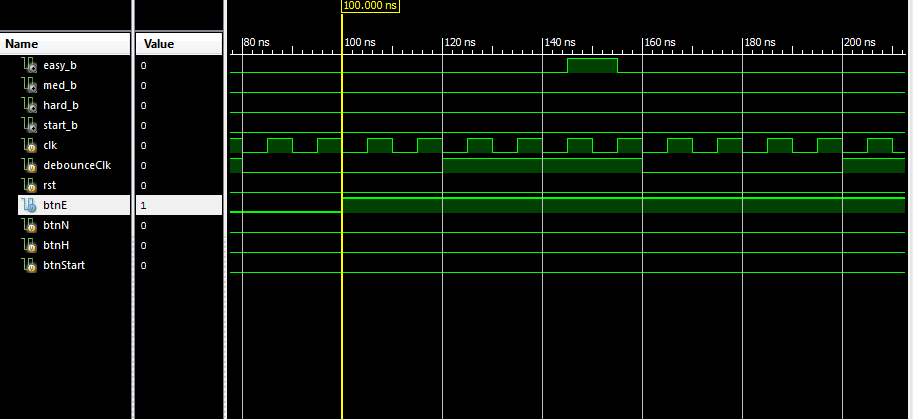
The next module we tested was our pseudo-random number generator, random.v, using the testbench rand\_tb.v. This module was used to generate the three numbers used in deciding which led’s would be active as the moles. Below we can see the waveform generated:



*Figure 4: Random number generator waveform*

We see that on every posedge clock a random, 3 bit number is generated, and the values of num1, num2, and num3 are updated accordingly (the parts in the waveform where a number lasts for more than one clock cycle are due to that number being generated multiple times in a row). We used online documentation to implement our own linear-feedback shift register (LFSR). While we know that the numbers are in theory not random, the actual LFSRs being generated by our module are 32 bits. These values are then modded to 8 bits and therefore the cycles they go through are much longer, and harder to recognize. In addition, the numbers are generated on the posedge of the base clock. Due to the different intervals of each level’s timer, this will result in a different set of numbers for different

The last thing we tested using the waveforms was our debouncer, using the testbench debounce\_tb.v. In this simulation, this testbench waited 100 ns, then activated the input btnE, and continued. We can see the produced waveforms below:



*Figure 4: Debounce testbench waveform*

Here we see that at the 100 ns mark, btnE flips to 1. Then, while the posedge of the debounce clock is active, the btnE input is sampled and the output easy\_b is set on and off to indicate the button was pressed. This shows the debouncer working as intended, and that it will properly clean out any bad signal seed in implementation.

Once these modules were tested individually, we were able to test somewhat through testbenches, however the most practical method at this stage would be through playtesting an implementation of the game. The main reason for this is that by playing the game, we are able to rapidly test how all the game’s systems interact with one another much more efficiently than through setting up multiple test benches.

**Conclusion**

Overall, this lab allowed us to have the independence to create a full Verilog project from start to finish with added elements of creativity. From the initial idea, to planning the implementation, to actually implementing different modules and functions, we were able to successfully make a Whack-A-Mole game on an FPGA board. We learned the process of iterative design, implementation, and testing, and used the tools that we needed to make it happen. We were able to isolate different functionalities and make everything work together by examining waveform diagrams and strategically designing modules for this purpose. One of the main challenges we encountered was with certain edge cases that required hardware/software synchronization. It was difficult to isolate which bugs were specifically hardware bugs versus which were caused by flaws of our logic. By utilizing testbenches for each module and waveforms to see where each signal was activated, we were able to solve these issues. Other challenges we encountered were with the initial approach in how we should go about structuring and making the Whack-A-Mole game. With collaboration and hard work, we were successful in getting it done and achieving all of the goals that we set forth in our initial proposal. Ultimately, we appreciated the level of independence of this lab, and thought that it could be improved with certain recommendations or libraries/techniques, such as a random number generator or other suggestions for implementing aspects of the game. All in all, this lab emphasized the design process in creating a fully functional Whack-A-Mole game from start to finish.