Whack-A-Mole Project Report

Simon Gilbert, Grayson Wiggins, Paul Adan, Will Holden, Yousuf Baker

Objective

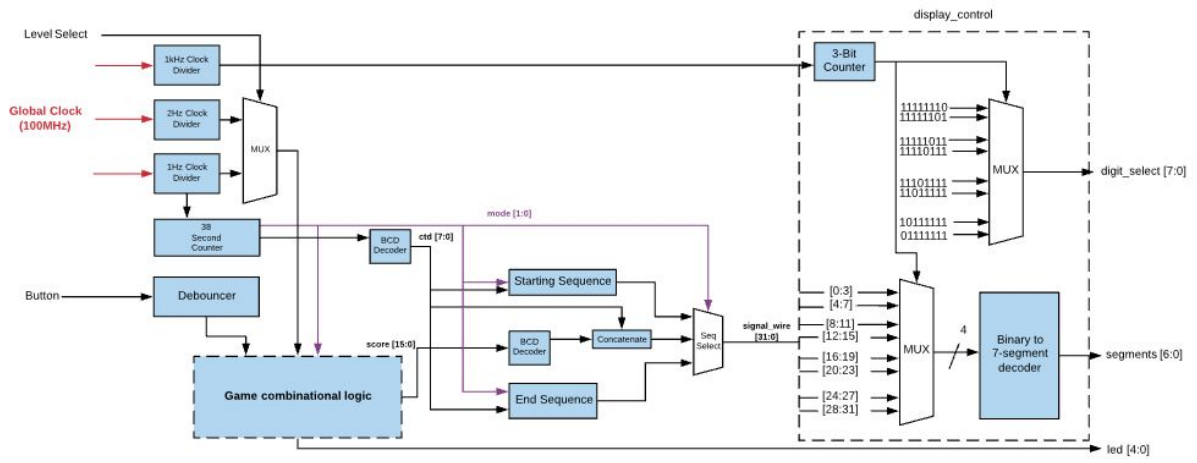
Our goal for this project was to build a playable Whack-A-Mole game in Verilog and implement it on the FPGA board. The “moles", represented by LEDs on the FPGA, get "whacked" by the user pressing a button corresponding to each of the LEDs. We use the 7-segment display to show the game mode-select, the game timer, and the user's score.

Methodology

Top:

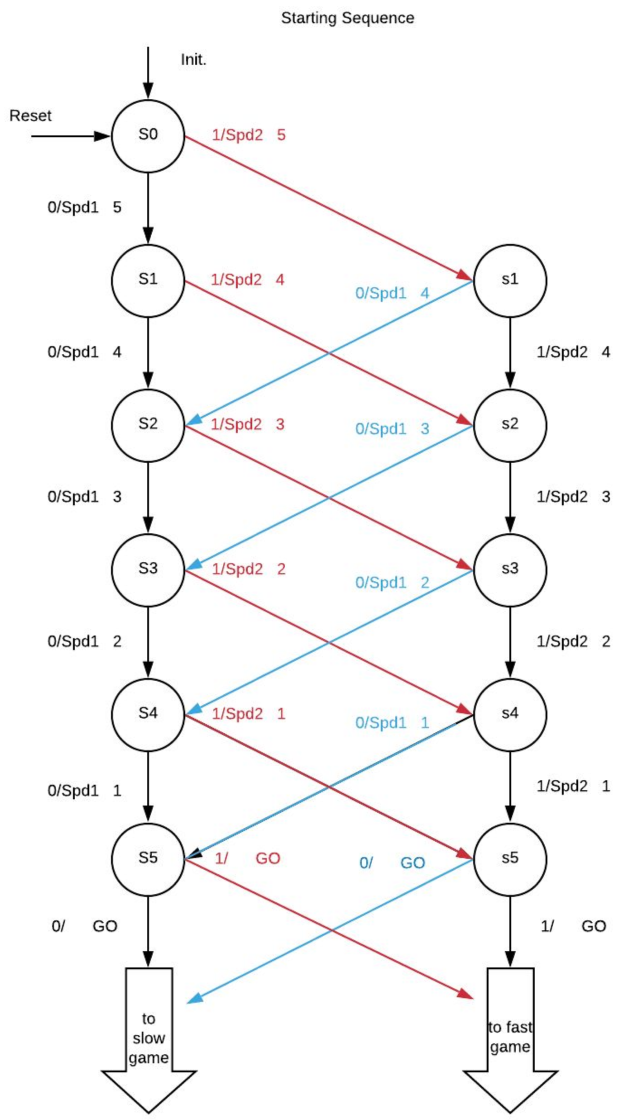
Our game features a starting sequence where the user has 6 seconds to select an easy or difficult game mode. During gameplay, one of five LEDs is randomly turned on at a speed determined by the selected difficulty level. If the user presses a button corresponding with the lit LED, their score increments by one. After 30 seconds of gameplay, the game ends, the final score flashes, and the came can be reset using a switch on the FPGA.

Figure 1: Top module diagram



We used a modular approach as shown in figure 1. The game has 3 basic components: a starting sequence, gameplay, and an ending sequence. All other modules are used to build up these components and control switching between them.

Figure 2: State machine for starting sequence



Starting Sequence:

The starting sequence is a Mealy state machine (shown right) that determines the display during the first 8 seconds before the gameplay starts. Depending on the state of the level select, it travels down the left column (level select = 0) or the right column (level select = 1) to the slow or fast game respectively. The transition time between states is dictated by the 1Hz clock. The user has the ability to switch levels for the duration of the starting sequence up until the transition into gameplay mode.

Multiplexers:

There are 2 multiplexers utilized in our design. The first is an 8-to-1 mux, the functionality of which is discussed in the display control section below. The second is a 2-to-1 mux which is controlled by the level-select switch and is used to select whether the 1Hz or 2Hz clock is forwarded to the game engine. The third mux, controlled by the 38 second timer, selects whether the starting sequence, gameplay, or end sequence is forwarded to the display control. More details for this mux are presented in the 38 second counter section below.

End Sequence:

The end sequence is a Moore state machine that begins once the gameplay mode ends (i.e. once the 38 second counter reaches 38 seconds.) It passes through 6 states (shown to the right in figure 3) at a rate determined by the 1Hz clock. The main purpose of this module is a e s t h e t i c, as all it does is flash the user's score on the LEDs while displaying "end" to indicate that the game is over. Once the user's score flashes three times, it will stay in state E5 until the system is reset using the switch on the FPGA. If in the event the system goes into an unknown state, it will default to a blank screen from which the user is indicated to use the system reset.

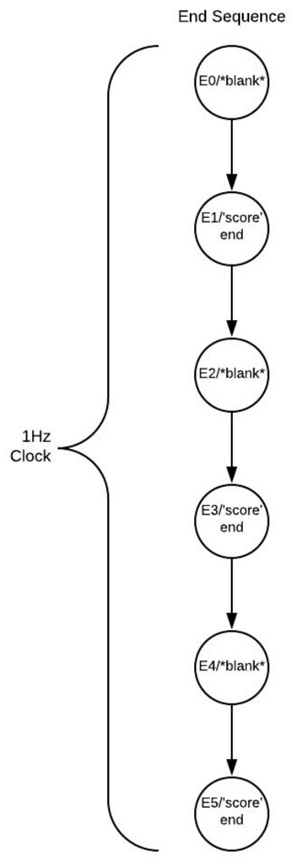


Figure 3: State machine for end sequence

38 Second Counter:

This counter controls the overall flow of a game. Its primary objective is as the select line to the mux that controls which of the game components (starting sequence, gameplay, or end sequence) is forwarded to the display control. The starting sequence is active for the first 8 seconds that the counter counts. The following 30 seconds activate the gameplay mode, and after that the end sequence is active until the user resets the game, at which point the counter begins again at zero.

The 38 second counter also acts as an enable to determine when each of the game components starts running. The final purpose of the 38 second counter is to act as an internal 1 Hz clock for the starting, gameplay, and end sequences as opposed to the universal 1 Hz clock.

Clock Dividers:

The three clock dividers (1Hz, 2Hz, and 1kHz) use the 100MHz global clock to output a slower clock. The clock dividers are implemented using a counter. The counter increments every global clock cycle and flips the slower output clock every time the counter reaches its max value. There are three clock dividers in this device: one that outputs a 1kHz clock, one that outputs a 1Hz clock, and another that outputs a 2Hz clock. The 1kHz clock is used to control the refresh-speed of the 7-segment display and the 1Hz and 2Hz clocks are used to dictate the game speed

Debouncer:

The global clock for the FPGA system operates at 100MHz. If the button input from the user was fed directly to the counter, then multiple button presses would be outputted by the counter due to the number of clock cycles undergone while the button is still held down. There would also be some issues with the fact that multiple clock cycles would pass while the button input was transitioning between 0 and 1 possibly leading to some undefined outputs. Since we would like to avoid this, our new best friend becomes the debouncer! This module makes it such that an increment is only fed to the counter after a certain number of clock cycles (I chose 1,500,000) pass while the button is held down.

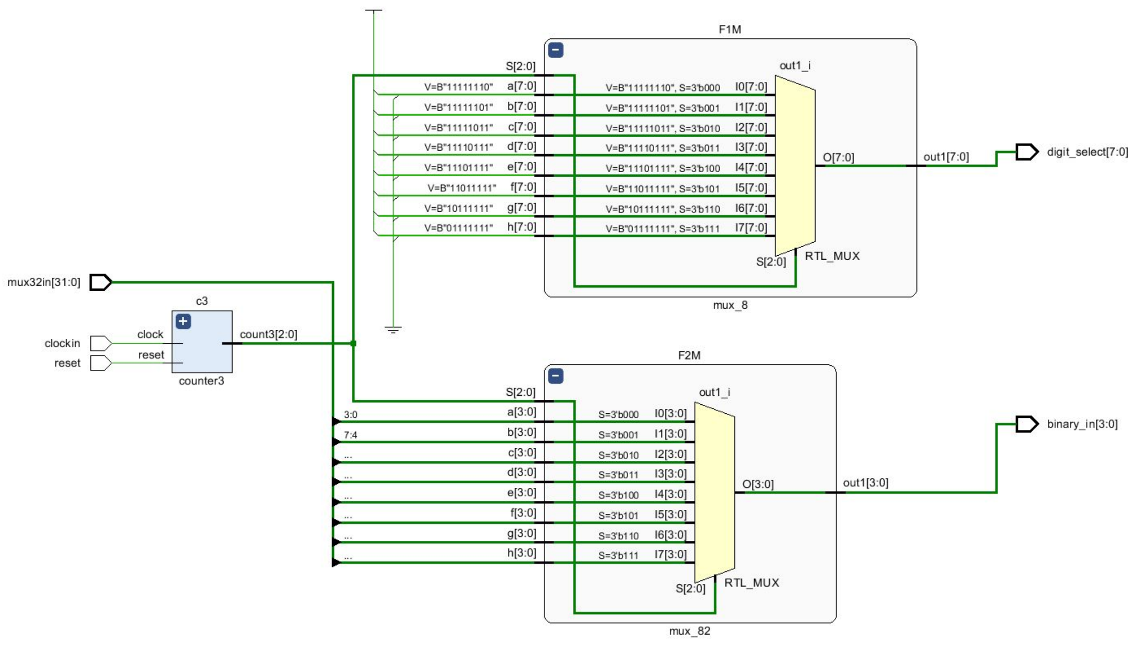
3-bit Counter:

This two bit counter creates a 3-bit output “select” that will rapidly cycle through the four inputs to the first instance of the eight-to-one multiplexer. The counter is triggered by the 1kHz clock so it will increment once every millisecond.

Display Control

The top-level module allows all eight displays to be shown at approximately the same time by using two multiplexers and a 3-bit counter. Since the FPGA board can only have one segment on at a time, the counter cycles at a high frequency of 1kHz and is fed into the 3-bit select of both multiplexers. One multiplexer serves as a digit select while the other outputs a selected 4-bit input into the seven-segment decoder. The fast cycle of selecting digits makes all eight displays appear to be on at the same time to the human eye.

Figure 4: Display control diagram



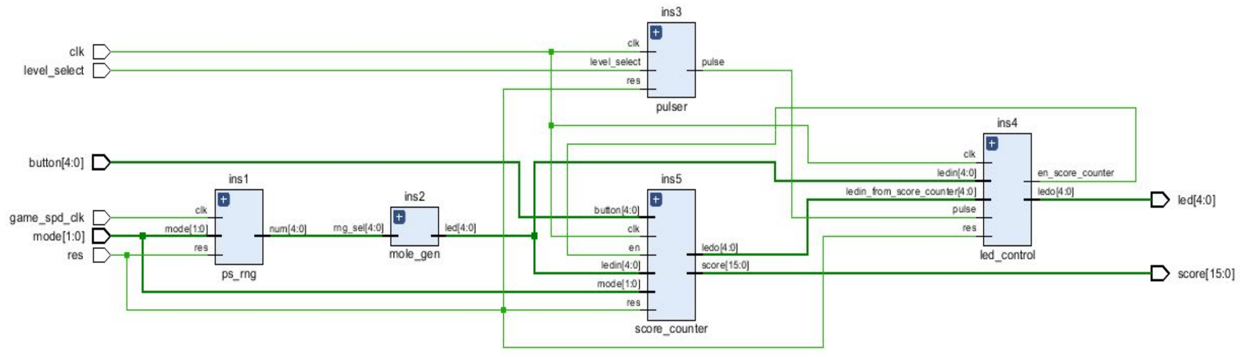
<u>Seven-Segment Decoder</u>

In order for the display control to be effective, it needs a seven-segment decoder to convert its output into a number or symbol. This module is purely combinational logic, and does not need a reset. Similar to Lab 2, the module implements 16 case statements and only needs a 4-bit input. Its output is a 7-bit number that determines which of the seven segments will be on in order to represent a number or symbol. Necessary characters produced by the decoder include the numbers 0-9, the letters in “Spd”, and the letters in “End”.

<u>Game engine:</u>

The game engine module is a top module for the collection of sub-modules responsible for creating the moles and keeping track of user input and score.

Figure 5: Game engine diagram



The game engine takes as input a divided clock, the main clock for an internal debouncer, the user inputted button (as a five bit bus corresponding to the five game buttons), and the mode and level select signals which control the rate at which moles appear depending on the level selected. The two outputs are the user’s current score and the led signal which turns on an LED corresponding to a mole. The game engine has two main components: the pseudo random number generator, and the collection of modules that comprise the score counter.

The pseudo random number generator is an augmented linear shift register (LSR). The most basic LSR is a device in which flip flops are connected in series and the bits are fed in and are outputted one by one at each posedge of clk. The RNG takes this basic design, and takes the output bits as in between the flip flops and at the output of the last flip flop and then passes them through XOR gates in order to produce the output bitstream. The LSR shifts and toggles each individual bit of the input to create a pseudo-randomized output, and 5 bits were chosen as the bus size in order to allow a sequence of 32 random numbers before the sequence repeats. Furthermore, the initial seed on which the LSR runs increments after every game to allow for 31 possible games. The increment was chosen to be 2'b10 to make it such that the seed is never 5'b00000, since a seed of 0 stays at a 0 steady state for all shifts.

The score counter keeps track of the user's score, and increases the score when a mole is whacked. The score counter also has an enable and only functions when enable is on. The button input from the user and the led (mole) input from the random number generator both use 5-bit one hot encoding. If the button input equals the led input from the random number generator, the score counter increases the score by one and the enable is switched to zero, disabling the score counter until the next 1Hz or 2Hz (depending on game speed) clock cycle. Every 1Hz or 2Hz clock cycle, the enable is set back to 1 by the pulser module. The pulser module is necessary because the enable variable in the score counter cannot be used in both always blocks.

The led control module turns off the led when the user whacks a mole. This is to show clearly to the user that they have whacked the mole.

Observations:

Sequence Specifications

In order to model this game as closely as possible to an actual video game, we had to carefully consider some explicit details. First, it is important that the user should not be able to increase their score during either the starting or ending sequences. For this reason an enable variable was instantiated within the score counter, preventing any illegal score increments. Once this improvement was implemented, it was clear that the game behaved much more like a commercially sold video game. Another way of improving the user experience was displaying the abbreviation “Spd#” during the starting sequence, to make the user aware of which level they are selecting without keeping track of pesky switches. Lastly, the blinking effect in the ending sequence was designed to mimic old-school arcade video games where lights would flash to attract users and maintain their interest. This could be observed as a tribute to the video game developers of old.

Game Engine

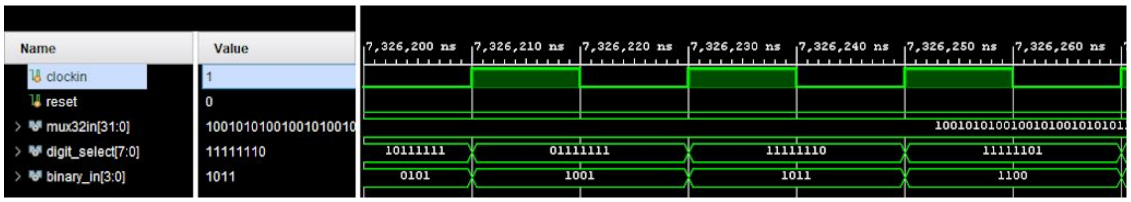
For the pseudo random number generator module of the game engine, it is not a truly random experience since the first time the game is run, the number sequence is always the same. Since the seed increments by the same value, it is possible for the player to memorize the game by memorizing all 31 games and their order. To add a second element of randomness, one could add a 5’bit counter running on 100MHz internal clock of the FPGA, from which the RNG samples a seed once the game starts. This way it is still possible for the player to memorize the games, but there is an extra element of randomness since the game and order of games is different every time. A key observation from the pseudo random number generator is that while the LSR serves this purpose, it does indeed follow a predictable pattern regardless of any augmentations. As such, it is easy to be memorized and is not an adequate option for any more serious designs that require robust security or cryptography.

We also observe a glitch in the game engine where the player would continue to receive points if the button last pressed matched the current led that was lit, even if the user only pressed the button 1 time. We fixed this by adding an enable to the score counter submodule, which turned off anytime the player received a point. The enable for the score counter turns on every time a new mole appears to allow the user to keep on whackin' moles.

<u>Display Control</u>

Looking at the testbench, the user can see that this module outputs the desired digit select and binary value because both multiplexers select the correct values. For example, when the digit select is '1111110', the binary_in is '1011'. This corresponds to the multiplexer selecting the rightmost display and 4-bit value from the input mux32in. In addition, these values are parallel and only update at the positive edge of the input clock, which in this case would be the 3-bit counter.

Figure 6: Testbench output for display control



Conclusion:

We learned that a modular design is best for efficiently testing each design component, assigning tasks earlier and equitably streamlines the design process, and that creating a user-friendly interface allows for the most enjoyable gameplay. Additionally, we learned that using synchronous clocks helps avoid entering unwanted states, that making testbenches before generating bitstream saves valuable time, and that it's best to use buses rather than single-bit inputs. We succeeded in creating a functional Whack-a-Mole game on the FPGA board, but the real treasure is the friends we made along the way.