EC311

<u>Whack-a-mole Final Project Report</u>

Team Members: Stanley Nguyen, Chia Jen Cheng,
John Sullivan, Kawsar Ahmed

Introduction:

For our final project, we designed a game of Whack-a-mole using Verilog, implemented on a Nexys-4 FPGA development board. Whack-a-mole typically consists of several holes, with moles popping up in them unpredictably. The objective is to “whack” the moles as they appear, earning a point for each successful whack. To recreate this classic game, we have five LEDs, which represent a mole by lighting up. Players press the push button that corresponds to the lit LED to score. In addition, players may change between two difficulty levels—the speed at which moles appear—by changing the position of a switch on the board.

The game can only be activated by pressing the reset button, which is followed by a five-second countdown. Scoring in the game is simple: pushing the button corresponding to the lit LED gives one point. Scores cannot increase by more than 1 at a time, and cannot decrease. A player’s current score is displayed as a decimal number on the rightmost four digits of the board’s seven-segment display. The remaining 4 digits show the time left in the game, with each game lasting 30 seconds. Moles are lit based on a pseudo-random number generator, to ensure no noticeable patterns emerge. This forces the player to compete based on skill, not simply recalling a sequence of button pushes.

Modules & Design

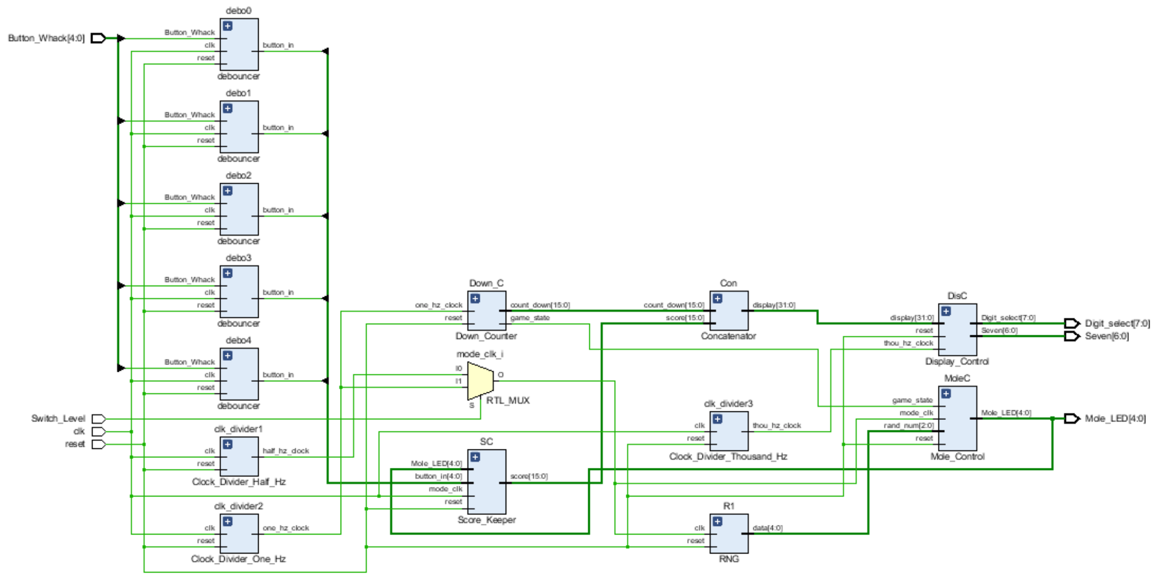

Figure 1*: Vivado Schematic*

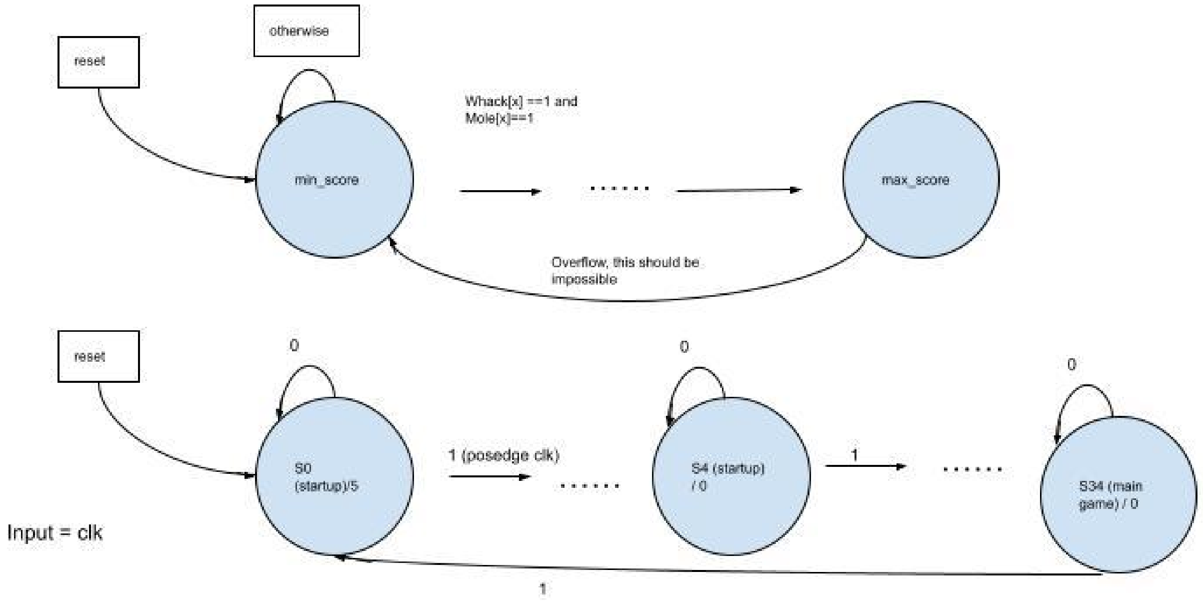

Figure 2*: State Diagram*

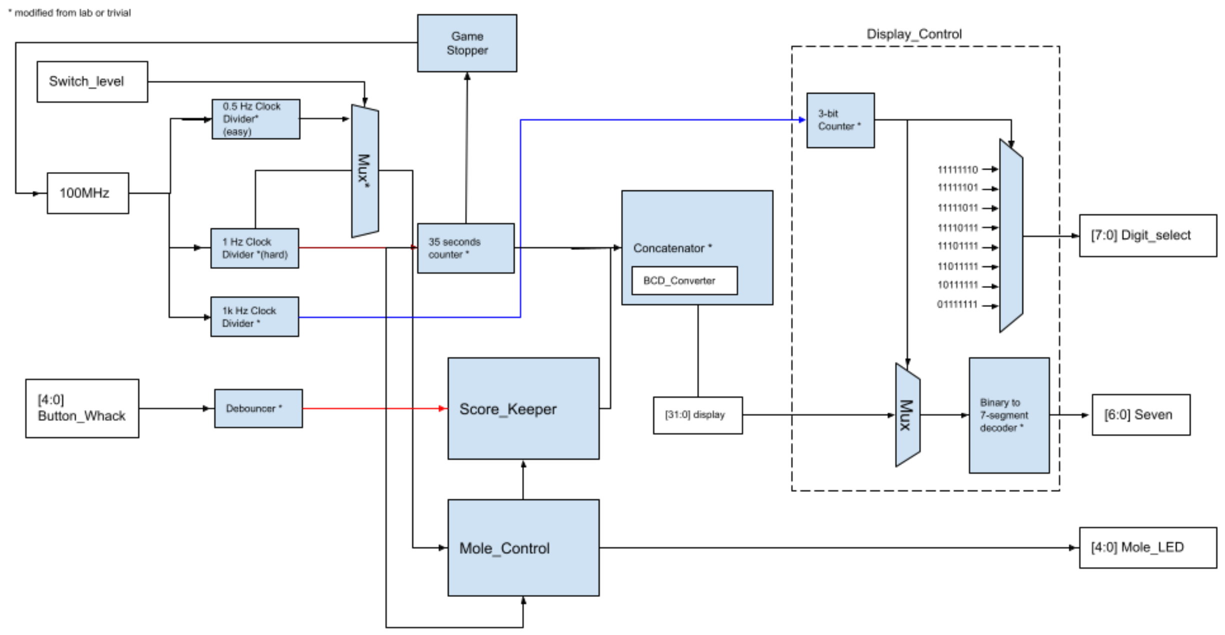

Figure 3*: Initial Schematic Concept/Planning*

To increase readability and for debugging purposes, the Verilog portion of the design is broken down into a hierarchy of submodules as shown in Figure 1.

Debouncer

The debouncer modules denoise the signals generated by push buttons to prevent the mechanical oscillations to be registered as multiple inputs. One debouncer module is dedicated to each button. They run off of the 100MHz clock, only passing through a button signal if their internal 21-bit counters are filled up. In other words, the debouncers require a steady button input for at least 1/[(100 MHz)/(2^21)] = 20ms.

RNG

This module generates pseudo-random binary numbers using a linear feedback shift register and a given random seed.

Score-Keeper

This module receives inputs from Mole_Control and all the debouncers. It would compare the one-hot encoded mole signal with concatenated inputs from the debouncers. If the two match, the score will increment by one. The score is stored as a 16-bit value, which is far in excess of the theoretical maximum. Practically, one mole per second on the hardest difficulty will only give ~30 points. The debouncers also put a hard upper limit of 1500 on the score (30 second game time, with debouncers requiring buttons to be pressed for at least 20ms). This can be seen in the first diagram of Fig. 2.

Clock-Divider

Multiple clock dividers are utilized to convert the built-in 100 MHz clock signal into three desired frequencies. Namely, 1 Hz, 0.5 Hz, and 1000 Hz. The game duration count-down uses the 1 Hz clock signal, and the mole pop-up periods are synchronized with either the 1 Hz or 0.5 Hz signal. Lastly, the 1000 Hz signal is required for the Display Control module for flashing the LED seven-segment display outputs.

Mole_Control

This module accepts a random three-bit binary number from RNG and converts it into a five-bit one-hot encoded number to generate moles. The frequency of the mole generation and removal is determined by the user-selected clock signal and an internal variable that disables the LEDs when the game is not in the 30-second countdown.

Concatenator

The Concatenator module receives the binary encoded score and countdown, converts them into binary coded decimal numbers, and combines them into an array. The numerical conversion step is crucial because BCD numbers are much easier to map into decimal numbers on the seven-segment display.

Display Control

This module is essentially the same as those created for the previous lab. The module takes in the desired value to display. Running at 1000 Hz, we activate the cathodes corresponding to the appropriate segments for each digit. Simultaneously, the anode corresponding to the digit in question is activated, turning that digit on, while the others are inactive. Because of the high clock speed, the digits all appear to be active simultaneously with different segments illuminated.

Down Counter

Each game starts with a 5-second countdown, followed by a 30-second game, as controlled by this module. When the 30 second game time runs out, the module changes the output of “game_state” from 1 to 0, indicating that the game is over. This output is given to other modules like Mole_Control and used to prevent the game from progressing. This can be seen in Fig. 2, with game_state=0 corresponding to S0-S4, and S34 in the second diagram. At the end of the game, the leftmost digits on the seven-segment display (which shows the time remaining in the game) will all read 0, while the rightmost four digits will continue to show the final score. This state cannot be exited unless the player uses the reset button to restart the game.

Functionality

Our design ultimately achieved the goals we had set out for it—the design functioned as intended, the game was playable, etc.

One practical observation was that the game is not particularly intuitive. The user needs to recall the mapping between the push buttons, which are arranged in a cross shape, and the mole lights, which are arranged in a line. During testing, we often took the shortcut of pressing all the buttons in quick succession to ensure we “scored” without having to memorize this mapping.

There were some flaws in the game design—in the real arcade game, a player hits the moles with a physical object, which imposes certain practical limitations on how quickly they can swing. Here, a player can easily press all 5 buttons before a mole disappears, even on the hardest difficulty. We proposed several solutions, including adding a cooldown to each button press, or increasing speed. Ultimately, we were satisfied with the difficulty of the game for now, and did not make those changes. Future designs would benefit from giving more thought to the implications of translating from a physical arcade game to a single PCB with lights and buttons.

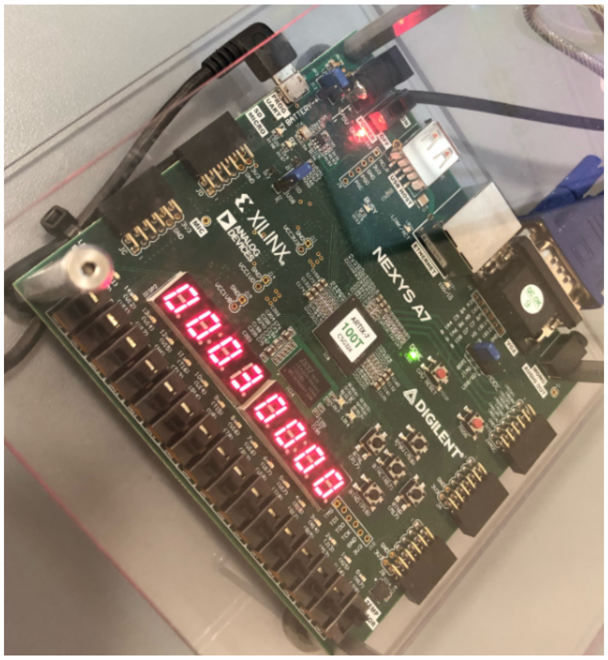

Figure 4: initial 5-second countdown to start the game

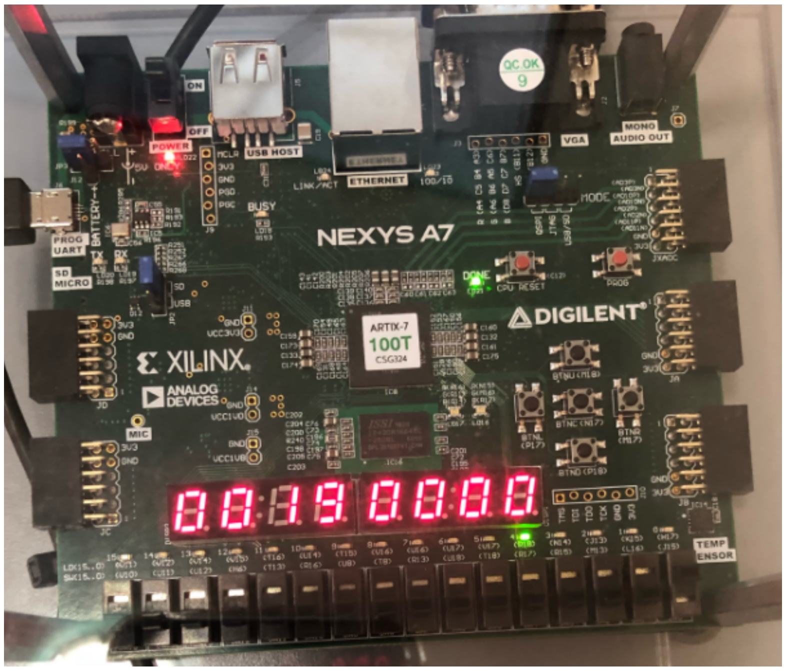

Figure 5: This is showing the in-game countdown while the moles are being generated randomly

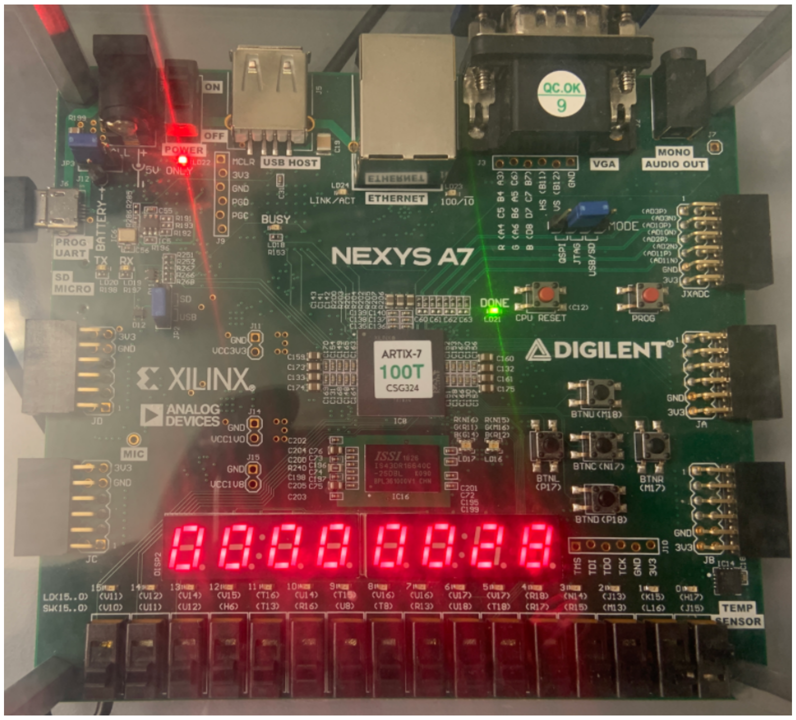

Figure 6: The game is over and it shows the final score of the game. It will reset once a new game has started.

Conclusions

While we achieved our functional requirements, it was not without some difficulty. In particular, difficulties with shutting down the mole LED after the mole is "whacked" took up quite a bit of time. At the root of the problem was that our Mole_Control module didn't have a functioning reset, due to small errors in the logic controlling the reset state. We missed this initially as our system-wide reset was functioning—this turned out to be other modules essentially masking the Mole_Control's missing reset. We only realized this once we observed that the mole LEDs would remain lit briefly after resetting, depending on the difficulty setting of the game. Since we turned off mole LEDs by triggering a reset of the Mole_Control module, the moles remained lit even after being pressed. Upon fixing that, the moles still remained lit. This ultimately was due to a foolish timing issue. We had the RNG running based on the same clock as Mole_Control (the 1Hz or 0.5Hz game clock, depending on difficulty). The result was that Mole_Control would re-sample the RNG, and get the same "random" number that it had previously. We fixed this by changing the RNG to run off the 100MHz clock.

Ultimately, we found that modifying the game on the fly, while temping during debugging, led to issues like this. We would have been far better served by proceeding carefully, updating our initial schematic, module input/output specs, etc, as we made changes. This would have forced us to think through things like which clock to use and where. We also would have been better off using git or another version control system, as we found on several occasions that we had caused an already-fixed bug to reoccur. Being able to see a detailed version history with commit messages would have made rolling the codebase back far easier.