Devan Dutta, Robert Griffith, Seokmin (Lewis) Hong
03/18/2018
CS M152A
Lab 4

Lab 4 Report

Introduction

Background

In this lab, we implemented a whack-a-mole game, using an FPGA Nexys 3 board and an attached PmodKYPD Numpad from Digilent. The 16 buttons of the numpad were mapped one-to-one with positions in a 4x4 grid that defined the game area. The game also increases in difficulty by way of level selection. As the user selects higher levels, the rate of appearance and disappearance of the moles is increased. Also, the placement of the moles is achieved through pseudo-random number generation in an LFSR algorithm, as recommended by our TA.

In order to complete this lab, we had to design individual modules to handle individual functionalities of the game and test those modules before producing a unified module. The most difficult module to test was the display module because even with very fine-grained testing of display signals in Simulation Mode, we cannot actually see what is happening without loading the code onto the Nexys 3 and looking at the monitor while the game is running. To minimize testing time on the FPGA, we wrote test benches for every module, including the display module, and confirmed via simulation that the behavior was working as expected. Secondly, we had to research the functionality of the Numpad before writing code to decode the numpad output. Third, we had to research how the VGA controller for the Nexys3 is implemented and used so that we could adapt it into a display module for our game.

Design Requirements

There were multiple functions that the game had to be able to perform. In order to provide a clearer depiction of the system, we will provide a game mode breakdown that includes some high-level implementation details:

Once the 4x4 game grid appears, and the score, timer, and level labels are visible, the user will select which level of the game to run by using the three right-most switches on the Nexys 3 board. Then the user can press the Nexys 3's left button (btnL) to start the game. The game will only begin if a valid level is selected, of which there are 3. To protect the input to the level selector module, logic had to be inserted that handled permutations of switches (more on that in the Modular Architecture section for level_selector). Secondly, to display the game, we had to implement a vga display logic controller.

As moles appear on the screen, the user must use the numpad to hit the respective positions of the moles. The mole positions are mapped one-to-one with the numpad keys when the numpad is in an upside-down position, so as to provide easier access to the buttons to the user (they don't have to place their hand over the FPGA and can rest their hand on the table). In order for the moles to appear pseudo-randomly on the screen, logic for creating and using an LFSR had to be implemented in the game. The appearance and disappearance of the moles is dictated by the level the user selected. Higher levels mean that the moles appear and disappear faster than they would have on lower levels. Additionally, to actually register that the user hit a mole, we had to not only research the workings of the Digilent PmodKYPD numpad, but we also had to map its buttons to mole states. Lastly, mole appearance happens by way of an animation, whereby three color arrays are used: mole is in hole, mole is at surface, mole out of hole.

The user has one minute to hit as many moles as possible. The timer and score label update as the game continues. So, the timer requires modulo counters for the tens and ones places. The score label required that we keep track of the number of times moles have been hit, and this is done in the mole module. Lastly, the level label stays fixed at the current level for 1 minute or until the user resets the game (more detail on resetting below).

The reset button is mapped to the right button on the Nexys 3 (btnR). The user can reset at the end of a level to simply reset the score and play again, or he/she can reset the game to an entirely different level, using both the 3 right-most switches and the reset button This is because a new level can only be selected once the player sets the switches to a non-zero value and presses the start button to start the selected level. The reset button will also trigger a re-draw through the vga display controller.

Figure 01 on the next page that shows gameplay during game execution.

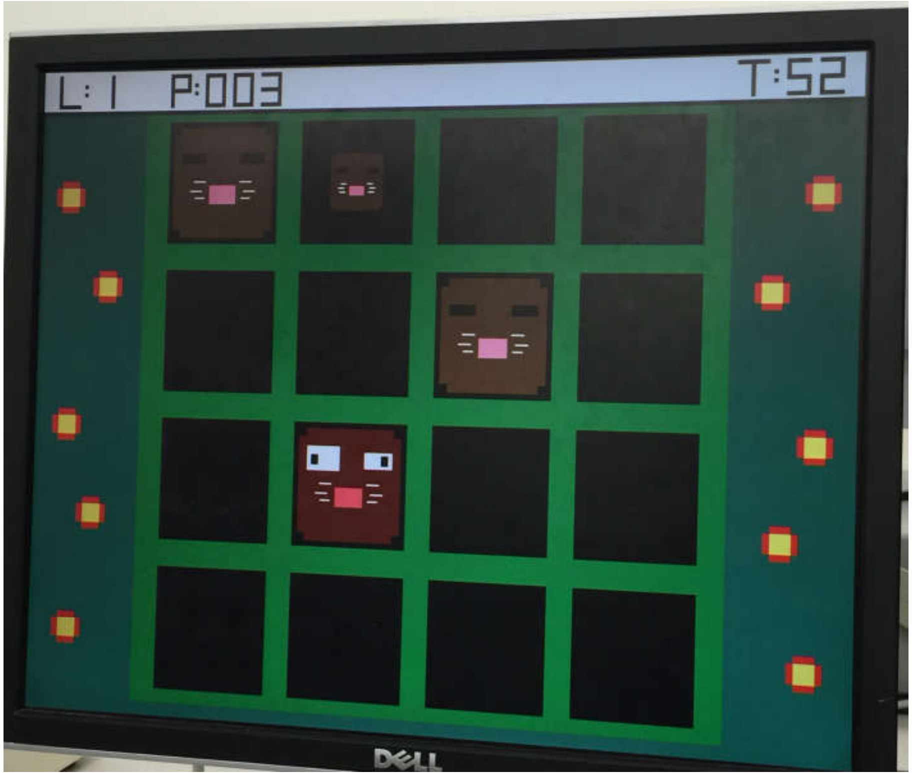


Figure 01: The above image shows a scene from gameplay. "L" indicates level, "P" indicates points and "T" indicates time remaining (in seconds).

Platform

This lab used both simulation and the Nexys 3 Spartan 6 FPGA. As individual modules were built, they were tested in simulation mode. Once unified modules were written, we could load the .bit file onto the Nexys 3 Spartan 6 FPGA to test the program in a more hands-on way. We limited the frequency with which we loaded the program onto the FPGA because it took time to generate the .bit file and program the FPGA each time we wanted to run a test.

A schematic and description of the system can be found in the whack_some_moles.v sub-section under the Interface of each Module section.

Design Description

Basic description of the design

To implement the game, we built 15 main modules. The first, called **bin_to_sev_seg**, converts binary encodings of single digits (0-9) into their seven segment equivalents. The second, called **clk_divider**, provides 4 main clocks: a 1 Hz clock, a 25 MHz clock, an animation_clock (used in each level to dictate mole appearance and disappearance), and an lfsr_clock (used to drive the LFSR module to come up with pseudo-randomized mole placement). The third, **debouncer**, serves to debounce the left and right buttons of the Nexys3, which are used for starting and resetting the game, respectively. The fourth,

display_vga640x480, is used for displaying the game to the monitor. The fifth, **level_selector**, is used to select which level of the game to run. The sixth, **LFSR_10**, is used to come up with pseudo-randomized numbers that are used for mole placement in the game grid. The seventh, **minute_timer**, is used for the minute timer in the game. The eighth, **mod_7_down_cntr**, is used for the tens place of the seconds displayed in the timer. The ninth, **mod_10_updown_cntr**, is used for the ones place of the seconds displayed in the timer. The tenth, **mole_object**, is used for controlling all mole state. The eleventh, **mole_vga_graphic**, is used for displaying a mole. The twelfth, **numpad_decode**, is used for just decoding which button was pressed. The thirteenth, **numpad**, maps the pressed button on the numpad to one of the 16 spots in the game grid. The fourteenth, **sixteen_moles**, is just a convenient wrapper for 16 moles that could be on the grid at various times. The fifteenth, **whack_some_moles**, regarded as the top module, is the module that ties everything together and uses the display controller to display the game.

Modular Architecture

The game is designed using 15 main modules, as described below:

The first module, **bin_to_sev_seg**, is responsible for converting a binary encoded single digit into its seven segment equivalent. It makes use of a 4-digit binary input and outputs a 7-bit seven-segment encoding of that input.

The second module, called **clk_divider**, provides 4 main clocks: a 1 Hz clock, a 25 MHz clock, an animation_clock (used in each level to dictate mole appearance and disappearance), and an lfsr_clock (used to drive the LFSR module to come up with pseudo-randomized mole placement). The clock divider makes use of the master clock, the reset signal, and the level, all as inputs. The four clocks described above are its outputs. Note that the animation clock makes use of the "lvl" (level) input because the clock becomes faster in subsequent levels.

The third module, **debouncer**, serves to debounce the left and right buttons of the Nexys3, which are used for starting and resetting the game, respectively.

The fourth module, **display_vga640x480**, is used for displaying the game to the monitor. This involves outputting a header with the level, score, and time remaining as headers, outputting a centered grid with 16 black squares and white columns and rows as delimiters between the squares, and outputting left and right borders so that the user's attention is focused on the center (where the game is happening). On a much more fundamental basis, the display module is controlling red (3 bits), green (3 bits), and blue (2 bits) coloring for the pixels that make up the game display.

The fifth module, **level_selector**, is used to select which level of the game to run. It takes in the switches as inputs and provides logic to protect the user input in selecting a level (for example, sw[1] = 1 and sw[0] = 0 should be level 2, but sw[1] = 1 and sw[0] = 1 can also be level 2).

The sixth module, **LFSR_10**, is used to come up with pseudo-randomized numbers that are used for mole placement in the game grid. Note that the LFSR algorithm is useful in providing only pseudo-randomized numbers, rather than true randomized numbers. We extended the base algorithm to work with 10 bits so as to generate a larger sample of pseudo-randomized bits. We then mapped the LFSR output to one of the 16 mole locations in the grid. Thus, the LFSR module is used each time we need to place a mole on the grid in a pseudo-randomized location.

The seventh module, **minute_timer**, is used for the minute timer in the game. The minute_timer is directly used in the whack_some_moles module for displaying the tens place for seconds and the ones place for seconds. In order to provide that output to the whack_some_moles module, the minute_timer makes use of the modulo 7 and modulo 10 counters, as described below.

The eighth module, **mod_7_down_cntr**, is used for the tens place of the seconds displayed in the timer. The minute_timer uses mod_7_down_cntr and then passes its output value to whack_some_moles for the "tens" variable.

The ninth module, **mod_10_updown_cntr**, is used for the ones place of the seconds displayed in the timer. The functionality of this module and where it is used is identical to what was described for the modulo 7 counter above, except for the fact that the modulo 10 counter is used for the ones place of the seconds, instead of the tens place.

The tenth module, **mole_object**, is used for controlling all mole state. This module uses the animation clock, as was introduced when discussing the clock divider, to control behavior such as: having the mole not being present (empty hole), having the mole barely in the hole, and having the mole outside of the hole. The mole_object state management also includes if the mole was hit by the user. Lastly, the mole_object module assigns points gained for the mole hit to the user.

The eleventh module, **mole_vga_graphic**, is used for displaying a mole. Where the previous module described was for maintaining mole state and behavior, this module is actually responsible for providing display information, including red, green, and blue data, to the display controller. To do this, the module has to take mole state and location into account.

The twelfth module, **numpad_decode**, is used for just decoding which button was pressed. In order to determine which button was pressed, this module is passed the row array for the numpad and the column array for the numpad. Then, it cycles through each of the columns, checking each of the four row buttons (there are 4 rows in a 4x4 grid). Starting from the top left of the numpad, the checking behavior is like this: start at top left column, go down all rows (checking to see if any of them were pressed), move up to top of next adjacent column to right, continue pattern. The numpad_decode module outputs a decoded_row and a decoded_col to be used in the numpad module.

The thirteenth module, **numpad**, maps the pressed button on the numpad to one of the 16 spots in the game grid. The module knows which button was pressed by making use of the numpad_decode module.

The fourteenth module, **sixteen_moles**, is just a convenient mapper for 16 moles that could be on the grid at various times. This module outputs mole states and point signals for the 16 mole objects it controls, each of which is controlled by a call to the mole_object module.

Lastly, the fifteenth module, **whack_some_moles**, is the module that ties everything together and uses the display controller to display the game. Briefly, that means debouncing, timing, numpad integration, mole behavior, score handling, and displaying. We will describe this module in more detail in the next section.

Interaction Among the Modules

All of the 14 other modules are used by the whack_some_moles module, which is effectively the main program for the whole game system.

First off, the whack_some_moles module makes use of the debouncer module for debouncing btnL (start) and btnR (reset). It passes the clk into the debouncer and it either passes in btnL or btnR for the debouncer's btn_raw, based on whether the call was for starting (btnL) or for resetting (btnR). The output of the debouncer module, called btn_state, is linked to the whack_some_moles.v's not_start and not_rst wires. These signals are then used to manipulate whack_some_moles.v's start and reset registers.

Secondly, the whack_some_moles module makes use of the level_selector by passing in the sw array to level_selector and binding sel_level to level_selector's selected_level.

Third, the whack_some_moles module makes use of the clk_divider, passing in the master clock, the reset signal, and the level. Then it uses the clock divider's outputs in the following wires: clk_1Hz, animation_clk, clk_25MHz, lfsr_clk.

Fourth, the whack_some_moles module makes use of the minute_timer to handle timer functionality. The minute_timer, in turn, using the modulo 7 and modulo 10 counters.

Fifth, the whack_some_moles module uses the LFSR_10 module to generate a mole start signal for a pseudo-randomly chosen mole.

Sixth, the whack_some_moles module uses the numpad module to see if a button was pressed. The numpad module, in turn, uses the numpad_decode module to determine which button was pressed and then uses that information (decoded row and decoded column) to determine which grid location was hit.

Seventh, the whack_some_moles module uses the sixteen_moles module to manage all the mole behaviors and states, including points awarded to the user. The sixteen_moles module, in turn, uses sixteen instances of the mole_object module to handle each of the behaviors and states for the 16 moles individually, and calculates the points scored by the player from the points stored in each mole_object module.

Eighth, the whack_some_moles module has to use the bin_to_sev_seg module for the hundreds, tens, and ones places of the score, the tens and ones of the timer, and the single digit of the level.

Lastly, the whack_some_moles module makes use of the display module. The pixel clock is 25 MHz, so that clock has to be supplied to the display module. The reset signal, the current status of the game (running/not running), the mole states, the score digits, the level digits, and the timer digits are also sent to the display module. Finally, the hsync (horizontal sync) and vsync (vertical sync) are sent to the display module. The outputs of the display module are bound to vgaRed, vgaGreen, and vgaBlue in the whack_some_moles module.

Interface of each Module

bin_to_sev_seg.v

This module is simple and utilizes purely combinational logic. Given a four bit binary number to the input four_bit_bin as a four bit wire array, the module will output the corresponding seven segment display encoding of the input value as a seven bit wire array named sev_seg.

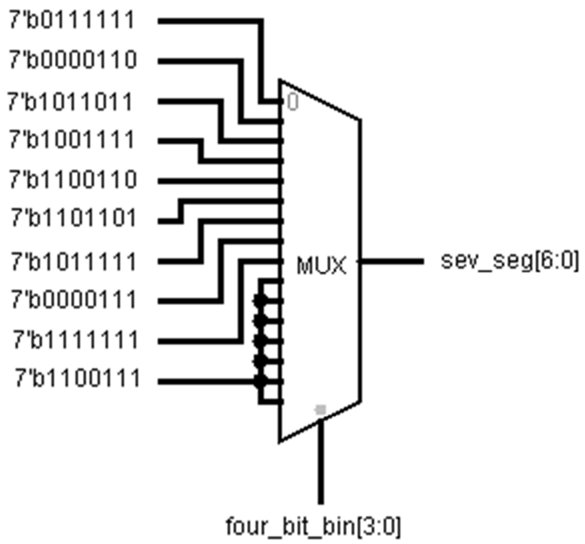


Figure 02: bin_to_sev_seg.v schematic. The image above diagrams the implementation of the binary to seven segment conversion module.

clk_divider.v

This clock divider module takes two single bit wire inputs: clk for the clock and rst for the reset signal. The module also takes a two bit wire array lvl that signifies the binary value of the current level. The general implementation of the clock dividers is to have registers for each counter increment given a clock tick from the 100MHz master clock that is passed in as clk. Once these registers reach a certain number they will right their clock high for a period of time and then fall back down. The module outputs four divided clock signals, each one bit. The 1 Hz and 25 MHz clocks were straightforward, as their frequencies do not change over the course of the level or with a level switch. The animation clock has a different condition for its incremental counter for each different level, speeding up from 10 Hz to 20 Hz to 30 Hz. The lfsr clock uses a shift register, as opposed to addition, to time its speed increase to occur once every three seconds. The shift register takes clk_1Hz as its clock, and the shift register holds three bits so by using non-blocking statements and initializing the register to 3’b001, every three seconds the lfsr clock will get faster. Once the register has a 1 in the most significant place, the limit to which the LFSR clock counter decrements.

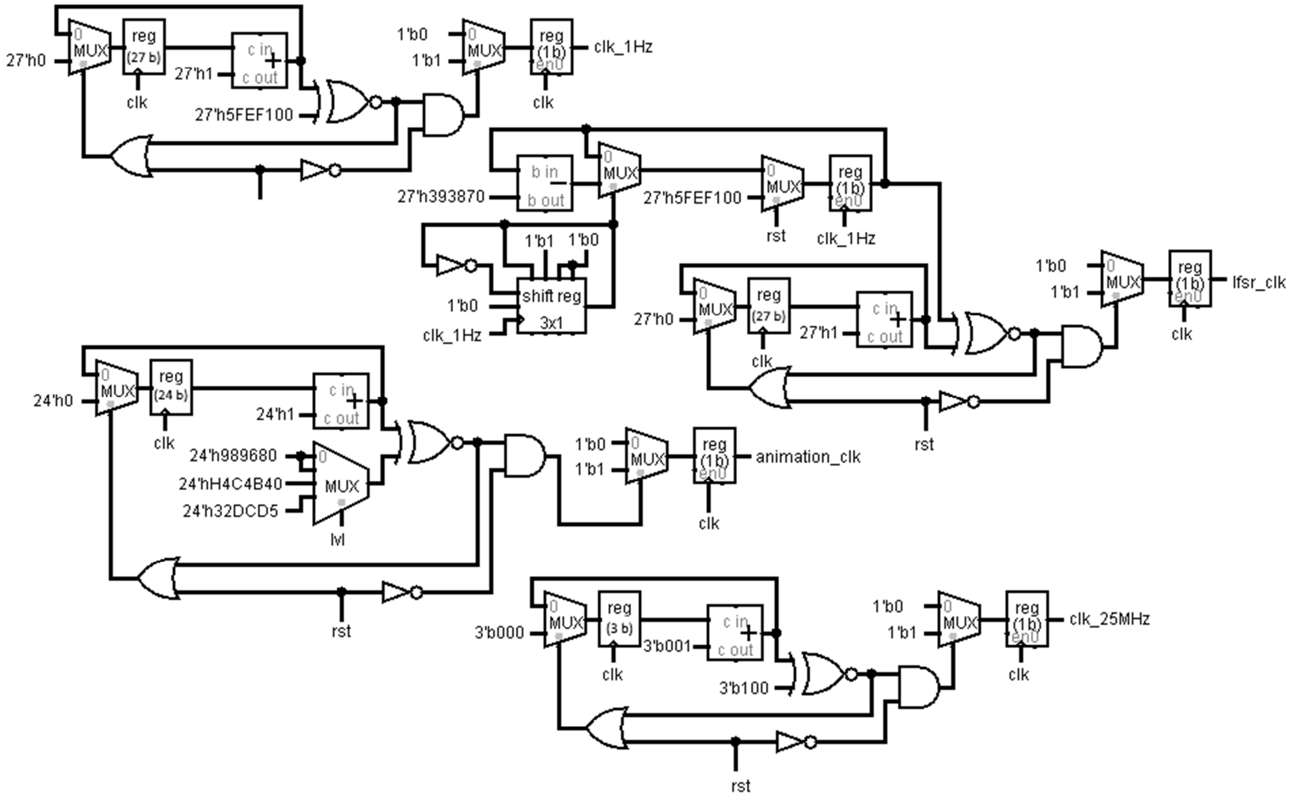


Figure 03: clk_divider.v schematic. The image above diagrams the implementation of the clock divider module.

debouncer.v

This module takes the 100 MHz clock (clk) and the wire connected to the button (btn_raw) as inputs, and outputs a register that is the debounced button state (btn_state). The implementation used for the debouncer uses two registers. The first is set equal to the button input, and the other is set equal to the this first register by a non-blocking assignment at each positive 100 MHz clock edge. If the current state of the button is equal to the value of the second register, a counter is reset to zero because the button is not in a different position. Otherwise, the counter is incremented. When this counter is full the state of the button is negated. The counter size is increased to increase the length of time a button needs to be not bouncing to register a change of state. By this method a button is debounced by not registering the rapid oscillations of a bouncing button as a change of state. The debouncer will only change a button's state after the oscillation has stopped and a new, steady value can be read for a certain period of time. We set this to about 5.24 ms, as we observed this to function well.

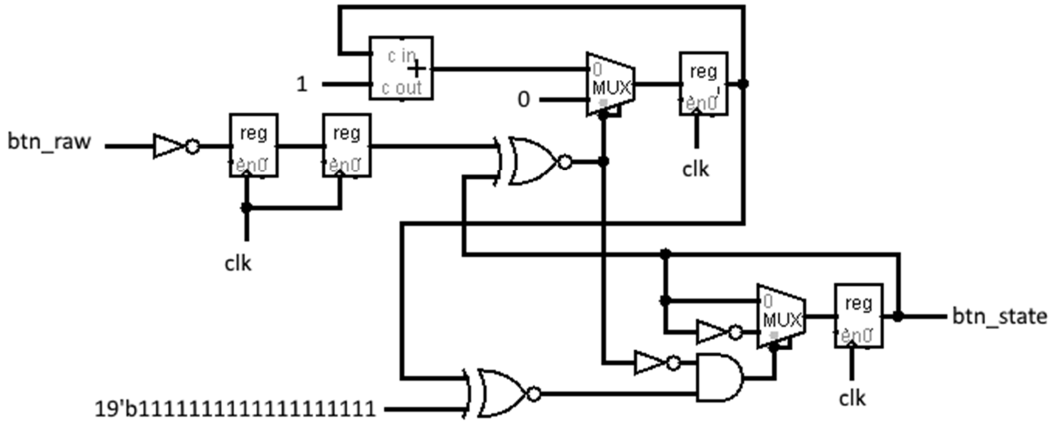


Figure 04: debouncer.v schematic. The image above diagrams the implementation of the debouncer module.

level_selector.v

This module is simple and utilizes purely combinational logic. The module takes a three bit input wire array (sw) and outputs a two bit wire array (selected_level), representing a binary level number. If the most significant bit of the input is high, the module outputs 2’b11. If the most significant bit is low and the middle bit is high the module outputs 2’b10. If both the most significant and middle bits are low and the least significant is high the module outputs 2’b00. Otherwise the module outputs 2’b00.

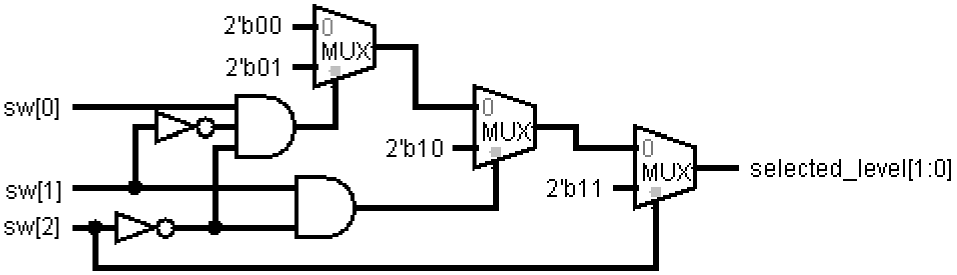


Figure 05: level_selector.v schematic. The image above diagrams the implementation of the level selector module.

mod_7_down_cntr.v

The modulo 7 down counter takes three input wires: a clear signal (clr), a clock (clk), and a count signal (cnt). The module outputs a terminal count signal and the value of the counter that is stored in a register. A four bit register, o_data, stores the counter output and is initialized to six. When given a count signal, the value stored in the counter register is decremented by one until it reaches zero, at which point if decremented the counter will reset to six by the next positive clock edge and immediately set the terminal count signal high, which is otherwise low. If the module does not receive a count signal it will not change the value stored in the counter register unless the module receives a clear signal. If the module receives a clear signal the counter register is set to six no matter what.

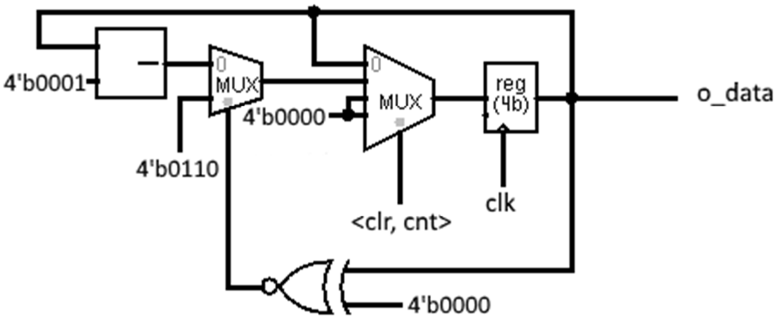


Figure 06: mod_7_down_cntr.v schematic. The image above diagrams the implementation of the modulo 7 down counter module.

mod_10_updown_cntr.v

The modulo ten down counter takes four input wires: a clear signal (clr), a clock (clk), a count signal (cnt), and an up signal (0 for down, 1 for up). The module outputs a 1-bit terminal count signal and the value of the counter that is stored in a register (4 bits called o_data). A four bit register, o_data, stores the counter output and is initialized to 0. The mod 10 counter's behavior is either incrementing or decrementing, based on the value of the up signal. If there is a clr signal of 1, then o_data will be reset to 0. If the module does not receive a count signal it will not change the value stored in the counter register unless the module receives a clear signal.

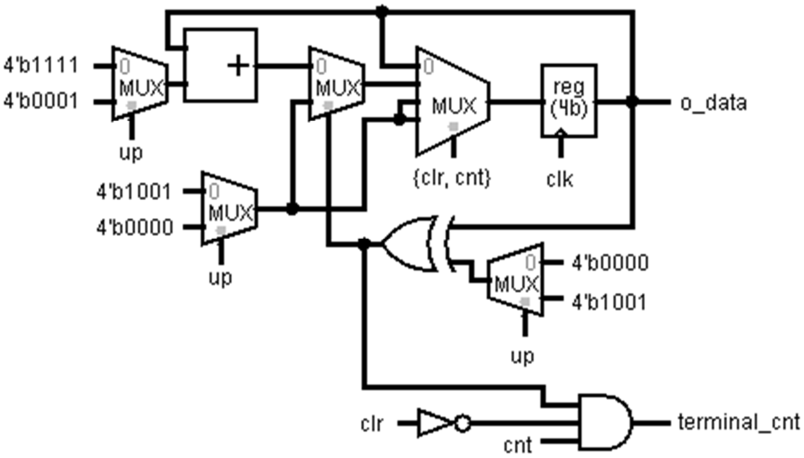


Figure 07: mod_10_updown_cntr.v schematic. The image above diagrams the implementation of the modulo 10 up-down counter module. Adding 4'b1111 to the four bit number stored in the register is equivalent to subtracting by one.

minute_timer.v

The minute timer module takes three single bit inputs: clr as a clear signal, cnt as a count signal, and clk as a clock signal. The module outputs two four bit wire arrays: the binary value of the tens digit on the timer (named tens) and the binary value of the ones digit of the timer (named ones). The module uses a modulo ten up-down counter to get the ones digit value, and passes clk as the clk to the counter, cnt as the count signal to the counter, and clr as the clear signal to the counter. The up input on the counter is constant low so as to count down. A modulo ten up-down counter is used similarly, with clr being used as the clear signal to the module, cnt used as the count signal, and the terminal_cnt from the modulo ten up-down counter is ANDed with cnt to be used as the count signal. The output of the modulo ten up-down counter is the tens output of the minute timer module.

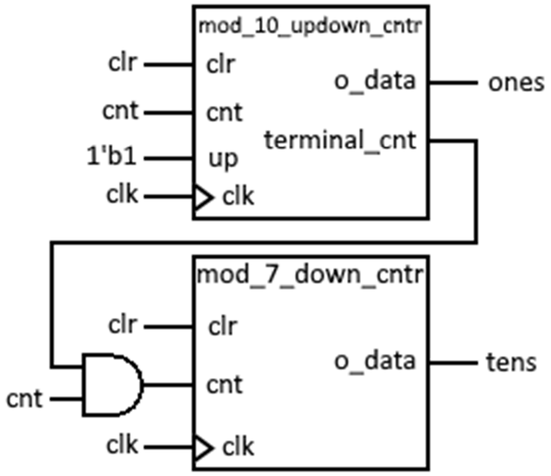


Figure 08: minute_timer.v schematic. The image above diagrams the implementation of the minute timer module.

LFSR_10.v

The lfsr_10 module takes in a 1-bit clk signal, which is supplied from the whack_some_moles module (lfsr_clock), and a 1-bit reset signal (rst_n). It outputs a 16-bit wire array (mole_location) that specifies where a mole should be placed in the 4x4 grid. The output is 16-bit because we are using an encoding scheme, whereby a 1 in a bit position indicates one of the squares and the rest of the bits are 0. The pseudo random number was generated using a 5 bit shift register. This would be initialized to 5'b11111 since 0 is a dead state and then we used XOR gates for the data shift register to find the data_next register. We took the data value and found the mod 16 value to get a hole location which was then encoded and set in the mole location array. Note that the LFSR algorithm only provides pseudo-random values, as opposed to truly random values. We used it as an approximation of random behavior in order to place our moles in the game grid.

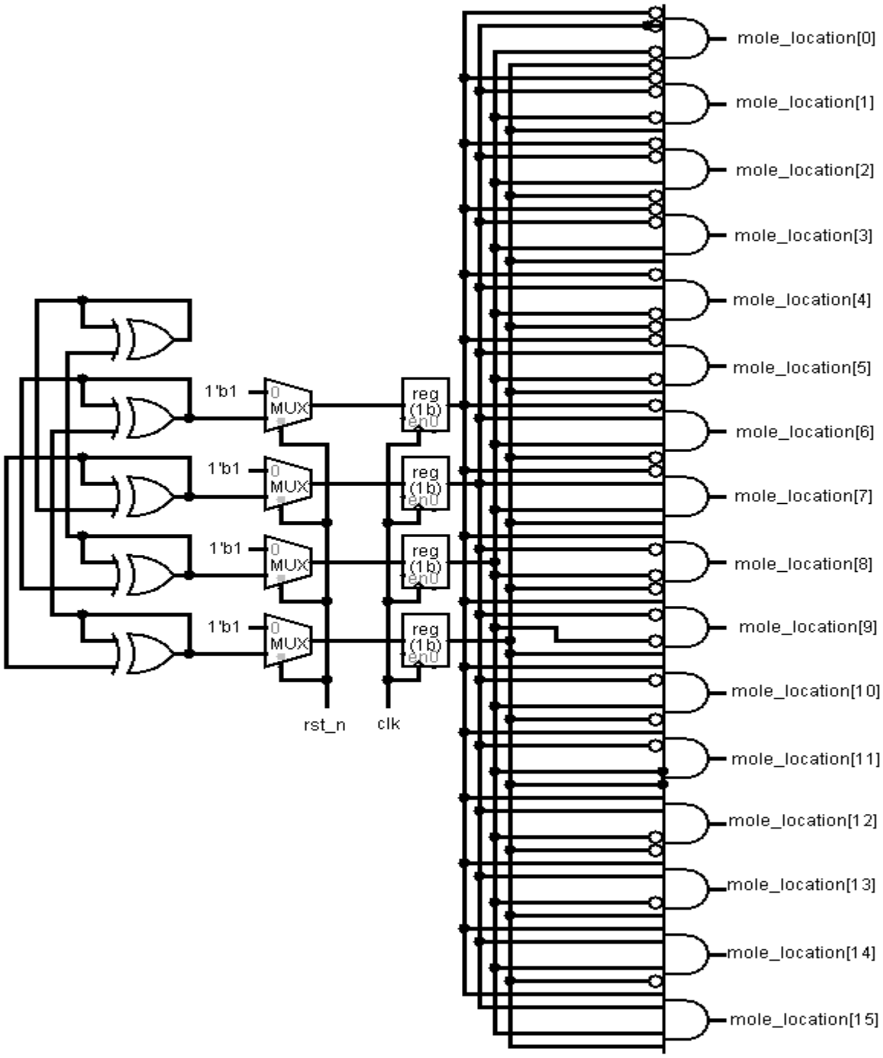


Figure 09: LFSR_10.v schematic. The image above diagrams the implementation of the LFSR module used to generate mole start signals.

numpad_decode.v

The numpad_decode module takes in a 1-bit clk signal (100 MHz master clock) and a 4-bit wire array for the rows of the numpad (called Rows). It outputs Cols (a 4-bit register for the column array state), decoded_row (a 2 bit register for which row was pressed), and a decoded_col (a 2 bit register for which col was pressed). Together, the decoded_row and decoded_col represent the pressing of a specific button on the numpad. They are also therefore a coordinate pair for our display grid. The module functions by setting the column value and then iterating through all the rows in the column to see if any of them were pressed. However, note that it uses reverse encoding logic (i.e. a 1 means “not signal high” and a 0 means “signal high”). Starting from the top left of the numpad, the checking behavior is like this: start at top left column, go down all rows (checking to see if any of them were pressed), move up to top of next adjacent column to right, continue pattern. The numpad_decode module outputs a decoded_row and a decoded_col to be used in the numpad module.

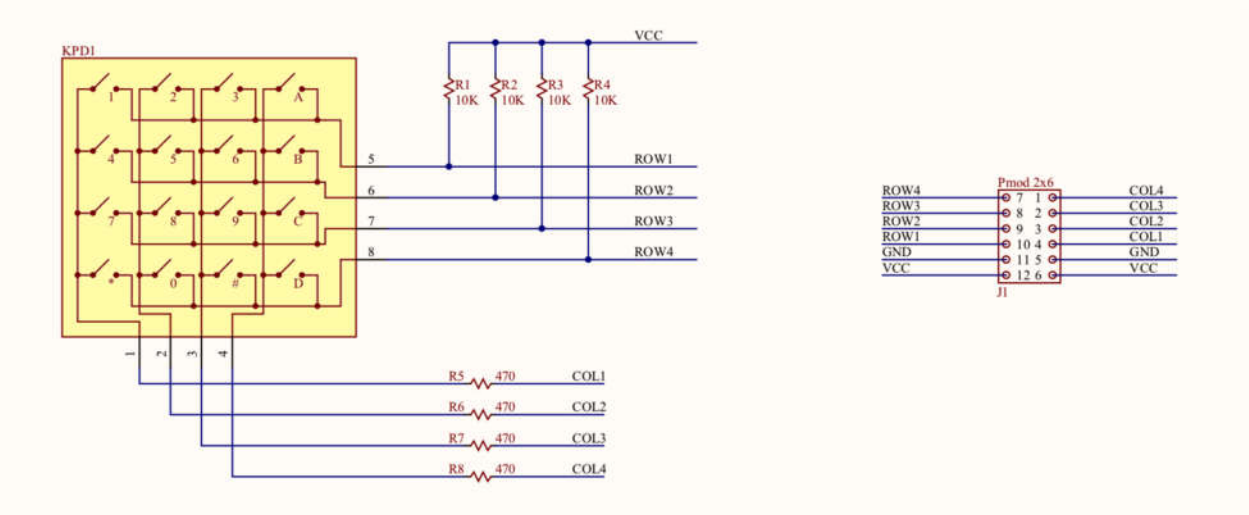


Figure 10: Schematic for the numpad. The image above diagrams the implementation of the numpad. The right side of the image shows how the rows and columns are mapped to the 12 pins that would be plugged into the JA port. The numpad module uses the JA as both input and output (as recommended by Digilent). Note that the bottom row is actually the buttons 0, F, E, and D, from left to right (the schematic still applies, but Digilent has had had multiple numpads). Source of image: https://store.digilentinc.com/pmod-kypd-16-button-keypad/

numpad.v

The numpad module takes in a 1-bit clk signal (the 100 MHz master clock) and an 8-bit wire array for the JA port (JA is the schematic name as represented in the Nexys 3 UCF for the port that we were using to plug in the numpad). The module outputs a 16-bit wire array that indicates which numpad button was pressed and therefore, which mole was hit (or if there were no mole, then just the grid location that was hit). The numpad module is a wrapper for the numpad_decode module in the sense that it calls the numpad_decode module in order to determine the decoded_row and the decoded_col. Then, the numpad module maps the decoded row and the decoded column to a grid location by way of a 16-bit encoding. For example, if the decoded_col is 0 and the decoded_row is also 0, then the output for the numpad module would be 16’b0000000000000001. Note that the last bit is 1 to indicate that the 0th spot in the grid was hit (because the grid has a 1-1 mapping with the numpad).

Lastly, note that the numpad module’s design is practically identical to the level_selector’s in the sense that a long ternary operator is implemented using AND gates and MUXes. The differences are that there would be 16 MUXes, one for each bit of the output, and the control signals to the MUXes would be from decoded_row and decoded_col.

mole_object.v

The mole object module takes five single bit wire inputs: animation_clk as a clock that dictates the speed of the mole changing states (its animation speed), rst as a reset signal, start as a start signal to make it possible for a mole to leave its hole, pause to hold all register values constant and pause the mole object, and btn as the signal for when the button corresponding to the mole is pressed. The module outputs the mole's state (named "state") which is stored in a two bit register, and outputs the number of times the mole has been hit (points_scored), which is stored in a six bit register. The logic is comprised of a series of many if conditions, all dictating what the proceeding value of state should be, and that value is evaluated and loaded into state at every posedge of the passed animation_clk. The logic implements careful checking of conditions when btn is high, as the mole should only be able to be hit in two of the states, when it is not retreating, and when its current animation period is not at its end (the expiration of the animation period supersedes the pressing of the button, making it slightly more difficult to hit the moles). The animation period for each state is stored in a four bit register named up_frames and the retreat status of the mole is stored in a one bit register named retreat.

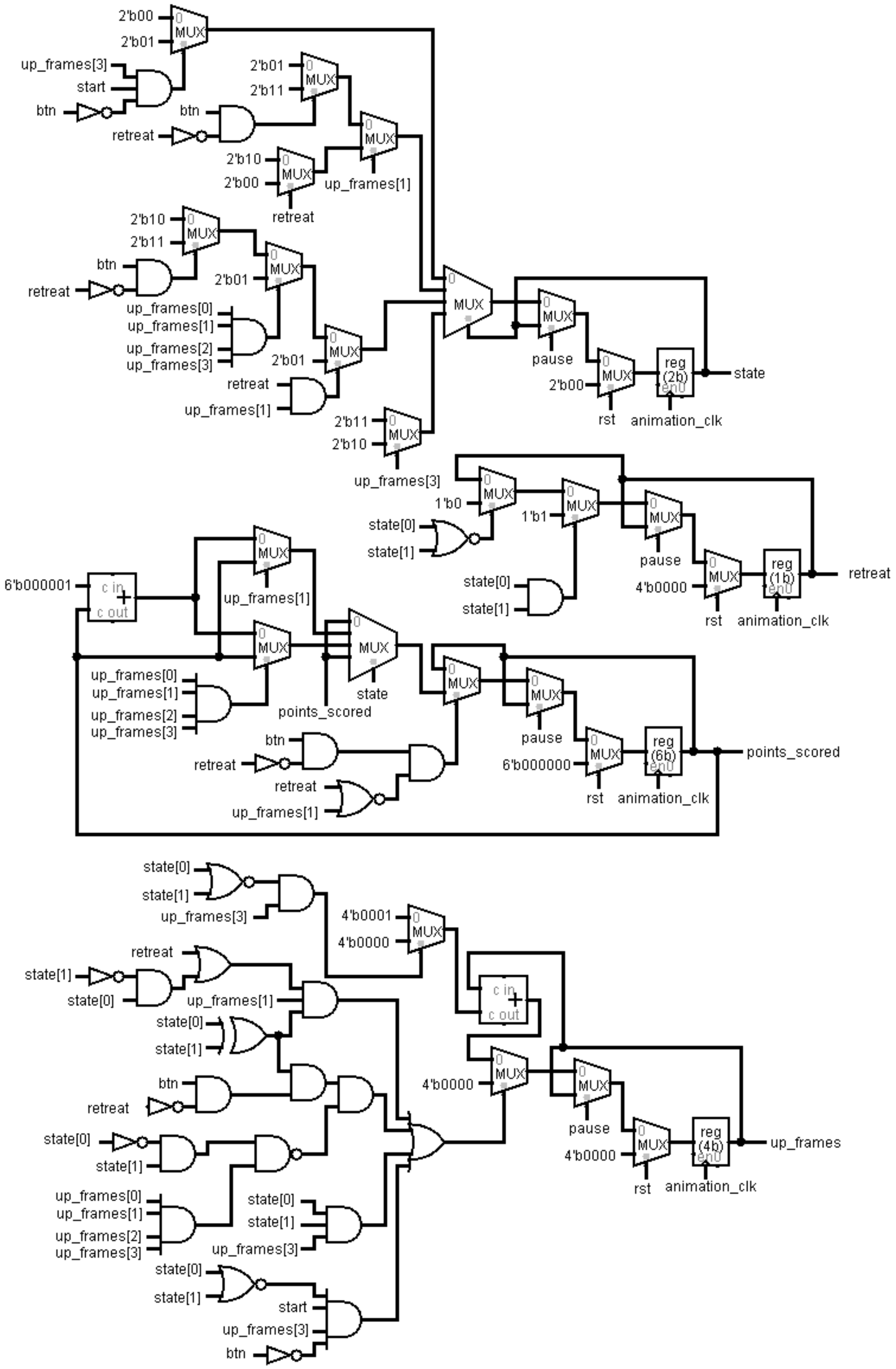


Figure 11: mole_object.v schematic. The image above diagrams the implementation of the mole object module.

sixteen_moles.v

This simple module is used as an organizational tool to keep the whack_some_moles module less cluttered. Note that sixteen_moles contains 16 mole object modules and takes three single bit inputs: animation_clk, rst, and pause; all of which will be passed to the ports of the same name in each mole object module. This module also takes two 16 bit wire array inputs, named start_moles and mole_btns. Each wire of start_moles contains a start signal to a specific mole object module, and similarly, each wire of mole_btns contains a btn signal to a specific mole object module. The sixteen_moles module ties these signals to their respective modules, and groups the states of all the moles into a 32 bit wire array output named mole_states. This module also sums the points_scored outputs of the mole objects and outputs three four bit wire arrays representing this sum: score_h as the hundreds digit, score_t as the tens digit, and score_o as the ones or units digit. These values are calculated using arithmetic that involves division.

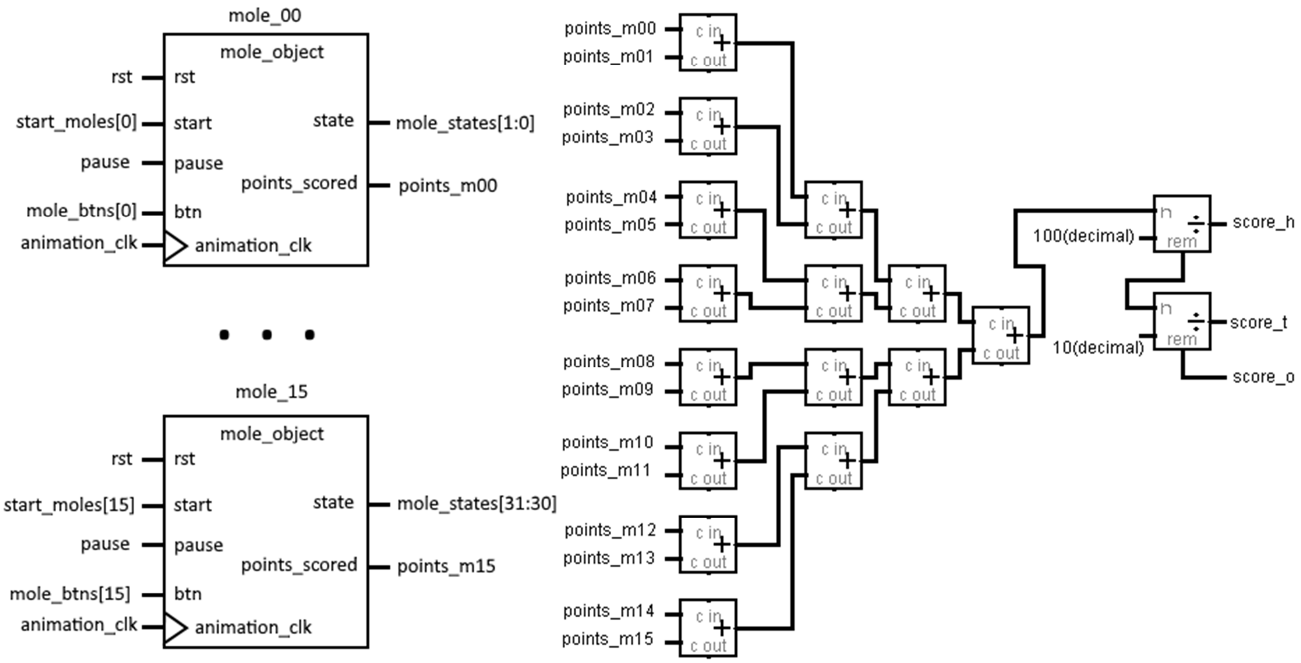


Figure 12: sixteen_moles.v schematic. The image above diagrams the implementation of the sixteen_moles module.

mole_vga_graphic.v

This module takes four input wire arrays: a two bit mole state (labeled mole_state), a ten bit row number (labeled rn), and a ten bit column number (labeled cn). The module produces a specific output for each set of values of rn and cn, which correspond to the row and column position of a pixel in the 90x90 frame of a mole. These outputs are three wire arrays: a three bit value for red, a three bit value for green, and a two bit value for blue. Different output is produced at different positions in order to draw a picture, and the module output at each pixel position varies for four different 90x90 frames; one for each mole state. While in Verilog the color values are stored in registers, this is for the sake of more readable code. The actual logic within this module is purely combinational. The code containing all if conditions needed to draw the graphics is hundreds of lines long, and diagramming each and every condition is extremely time consuming (even more so than the other diagrams in this section) unnecessary when the method can be modeled with a smaller example of producing the output for one region of the mole graphic.

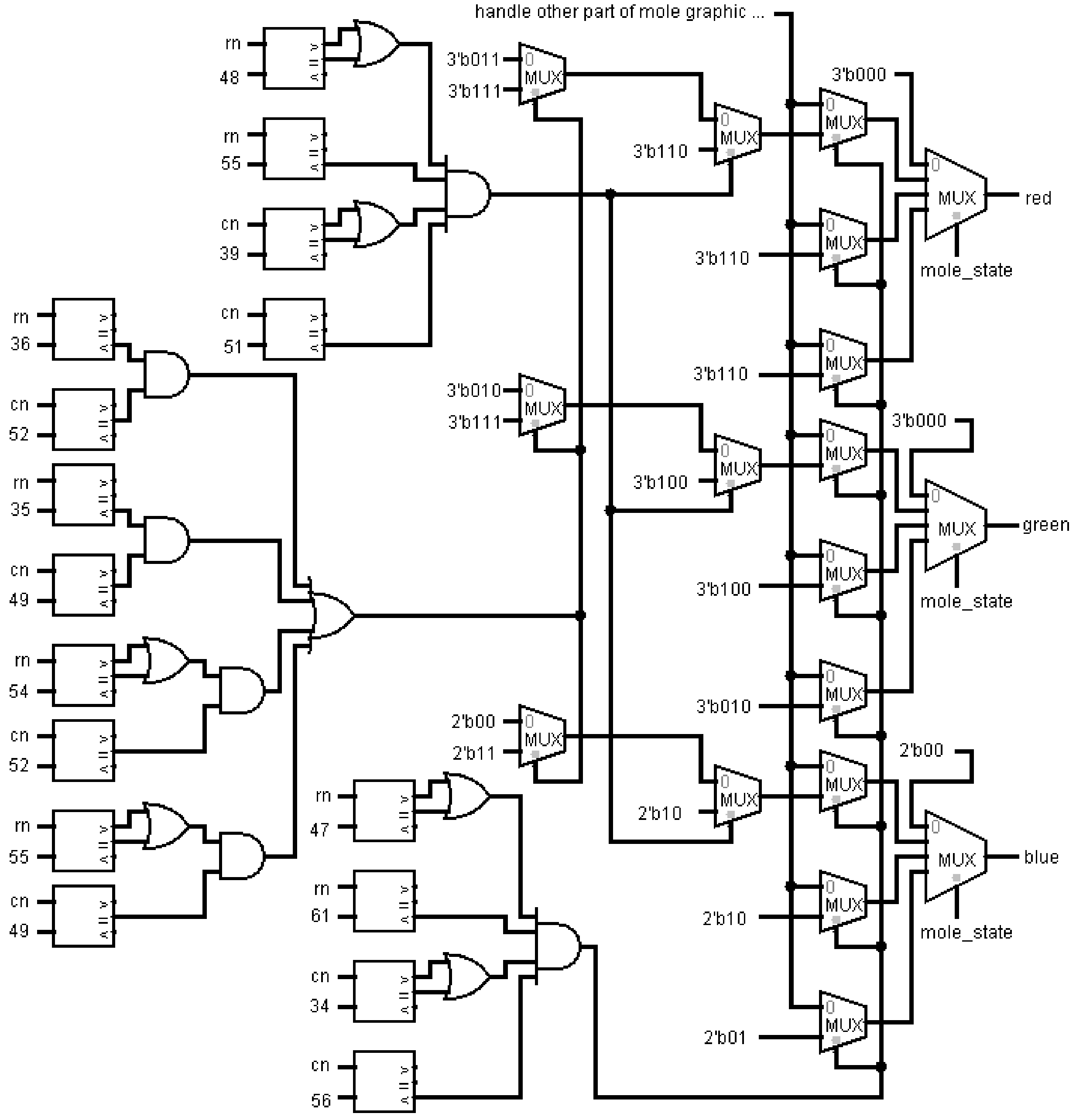


Figure 13: Partial mole_vga_graphic.v schematic. The image above diagrams the general principles behind the implementation of the mole graphic module by demonstrating the way the region containing the mole's nose is drawn.

display_vga640x480.v

The display module takes nine inputs, a single bit pixel clock labeled dclk, a single reset wire labeled clr, a 32 bit wire array of mole states labeled mole_states, and six seven bit wire arrays that are seven segment signals for the units digit of the score, tens digit of the score, hundreds digit of the score, level number, tens digit of the timer, and the ones digit of the timer. Respectively, these seven segment signals are labeled score_o, score_t, score_h, level, timer_t, and timer_o. The module outputs VGA signals, a single bit horizontal sync labeled hsync, a single bit vertical sync labeled vsync, and a three bit wire array for red, a three bit wire array for green, and a two bit wire array for blue. The color outputs are labeled simply red, green, and blue. The module iterates through each pixel in the 640x480 range of the VGA display by using registers for vertical position and horizontal position. There is logic that handles the colors to be output at each position. This is almost entirely done with comparators, as the module checks if the current pixel position is within a set of ranges and sets red, green, and blue according to that condition. The display is sensitive to the given inputs. The seven segment encoded numbers passed as input will be displayed in the header of the display by treating each segment as a range in which to output black if and only if the seven segment signal bit is high for that segment. This is applied to produce the level, score, and timer numbers on the display. The letters are displayed by simply setting red, green, and blue to zero when within specific ranges that are part of the letters.

The display is partitioned into a header, right banner, left banner, footer, 16 mole holes, and the grid which is the space between and 20 pixels around the mole holes. For the sake of themed levels, the banners, footer, and grid sections of the display change according to the level input passed as well, and this is achieved with multiplexers and XNOR gates to check for equivalency. The moles are displayed by using a mole_vga_graphic module, and passing it the value of vc and hc subtracted by the offset of the top left corner of the mole display region in the grid. The mole_vga_graphic module is passed the two bit portion of the mole_states input wire array that corresponds to the current mole hole. This way, only one mole_vga_graphic module is needed to handle the display of all sixteen moles. At the end of the module, depending on if hc and vc represents a pixel in a mole region, the module either outputs the red, green, and blue signals returned by the mole vga graphic module or the display modules own logic, which leaves its register values unchanged from the region before the pixel coordinates were inside the mole hole. The registers were distinguished with the color names followed by _bgrnd or _mole. Like the mole graphic module, the code containing all if conditions needed to draw the graphics is hundreds of lines long, and diagramming each and every condition is extremely time consuming (even more so than the other diagrams in this section) unnecessary when the method can be modeled with a smaller example of producing the output for one region of the display graphic.

Since the general principle of checking boundaries and outputting correct values for that region was demonstrated in the mole vga graphic figure, the below figure diagrams the way the seven segment display of digits over VGA was implemented.

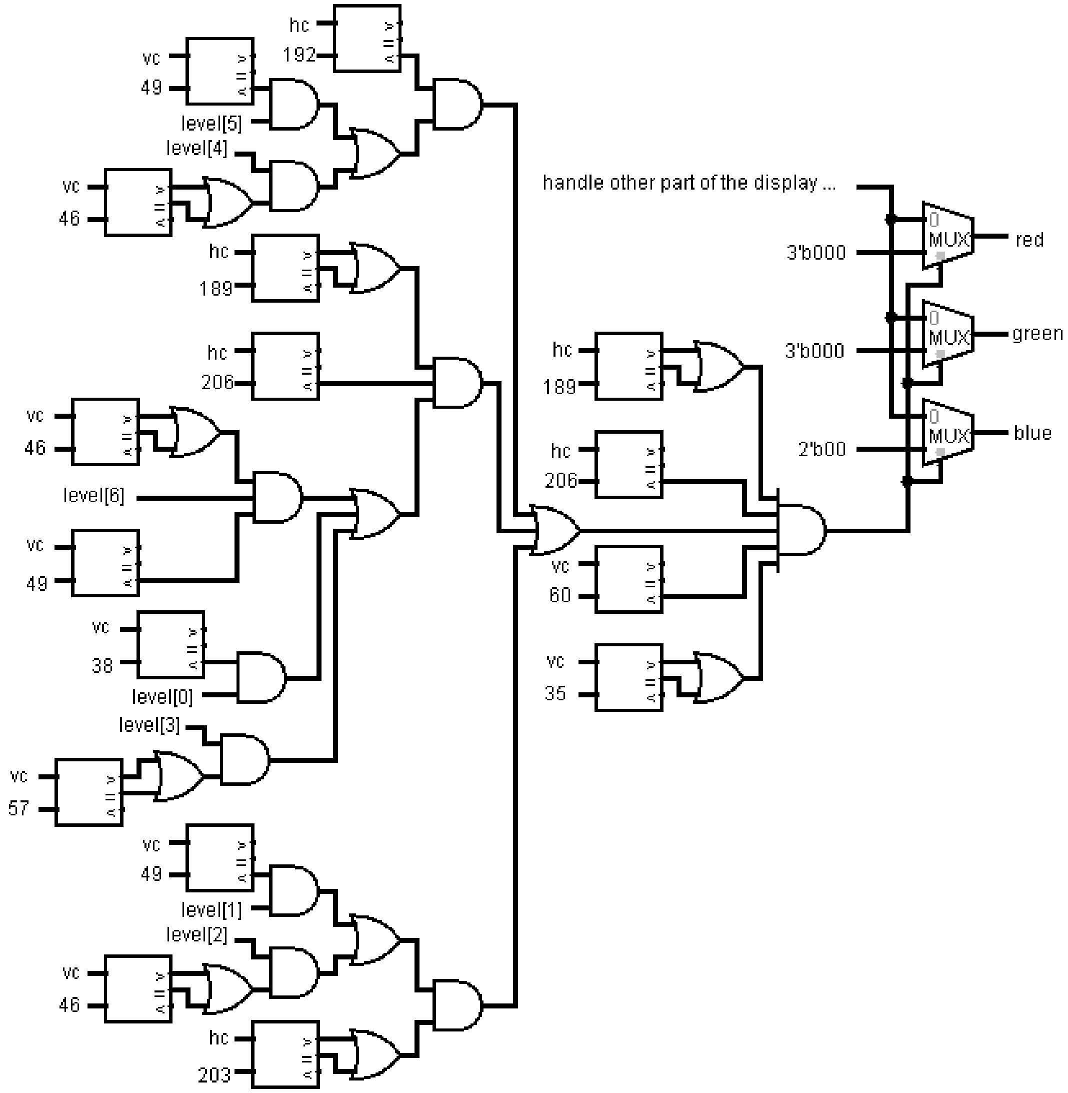


Figure 14: Partial display_vga640x480.v schematic. The image above diagrams the general principle behind the implementation of displayed numbers, using the level number as an example.

whack_some_moles.v

The whack_some_moles module is the top module. It takes in clk (1-bit wire for the 100 MHz master clock), sw (3 bit wire array representing the switches), btnL (a 1 bit wire for the left button), btnR (a 1 bit wire for the right button), and JA (an 8-bit wire array for the JA port of the Nexys 3). Note that JA is actually inout, as recommended by Digilent. The module outputs vgaRed (a 3 bit wire array for red coloring), vgaGreen (a 3 bit wire array for green coloring), vgaBlue (a 2 bit wire array for blue coloring), Hsync (a 1 bit wire for horizontal sync signal), and Vsync (a 1 bit wire for vertical sync signal). btnL and btnR are debounced by two individual debouncer modules, and the module uses their inverted output as the debounced start and reset signals respectively. The module passes the switch input sw into a level_selector module. A register storing the current value of the level is only set to this selected level whenever the game's run state, which is a single bit register, is low, start is high, and the switch input does not correspond to level 0. In these same conditions the game's run state is set high. A wire that controls the actual reset of the game, called game_rst, is high whenever the run state is low and start is high, or whenever the debounced reset signal is high. The top module passes the 100MHz clock into a clk_divider module to get four new clock signals. A timer module is passed the game_rst signal, 1Hz clock and game pause signal to the clr, clk, and cnt ports respectively. The game pause signal is a wire that is high whenever the run game state is low or when the timer reaches 00. This wire serves to pause the game when the player runs out of time, freezing the display so they can see their final score and the level as opposed to simply resetting. The player cannot unpause the game without first pressing btnR and then selecting a level with the switches and pressing btnL to start the game. The timer module outputs are fed to bin_to_sev_seg modules to be converted to seven segment signals and passed as inputs to the display_vga640x480 module. A 16 bit wire array labeled mole_signals is created from an LFSR_10 module, which is passed the lfsr_clk signal and the game pause signal inverted as rst_n, so that when the game is not paused it generates new numbers. A numpad module takes the 100MHz clock as input along with the JA pins, and outputs the 16 bit wire array with signals corresponding to each mole button. The moles are handled by the sixteen_moles module being used. The animation_clk is passed as input, game_rst is passed to the rst port, game_pause is passed to the pause port, and mole_signals is used to start the moles along with mole_btns being passed to register mole hits. The module outputs mole states and the score digits which are then converted. by bin_to_sev_seg modules. The level register has its value passed to a bin_to_sev_seg module as well. These seven segment signals along with mole_states, game_rst and the 25MHz clock signal are passed to the display_vga640x480 module which then produces the outputs for Hsync, Vsync, vgaRed[2:0], vgaGreen[2:0], and vgaBlue[1:0], which the whack_some_moles module outputs.

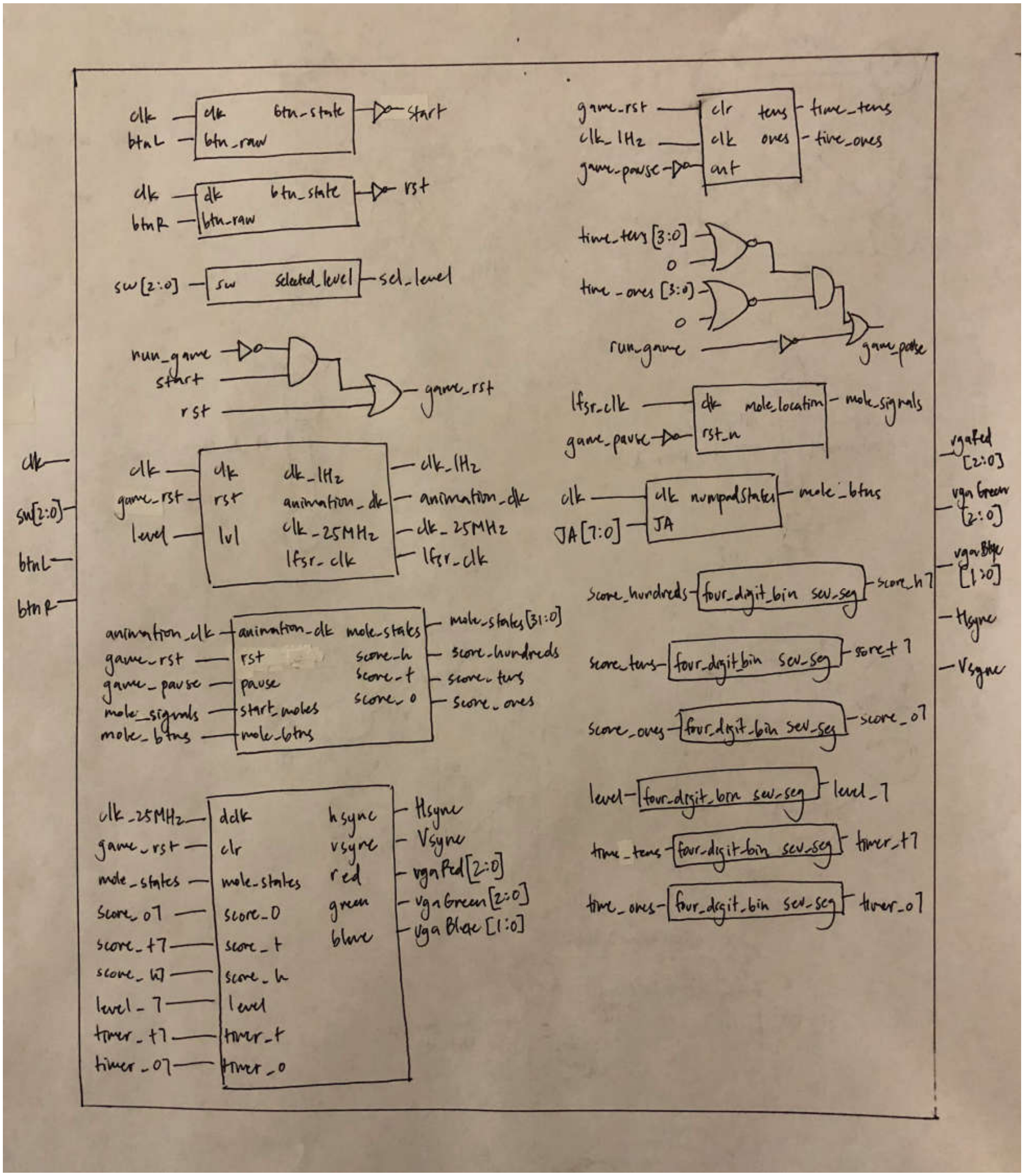


Figure 15: whack_some_moles.v schematic. The image above diagrams the implementation of the top module whack_some_moles.v. This module utilizes all previous modules described.

State Diagrams

clk_divider.v

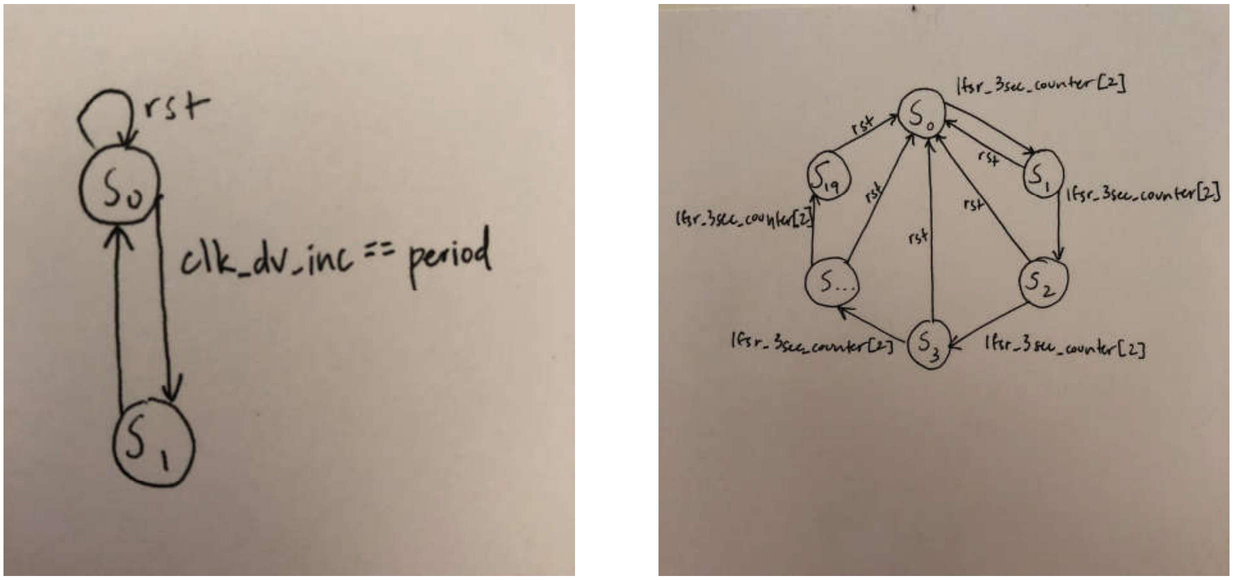


Figure 16: clk_divider.v The clock divider is a little different for each clock but each clock follows the same idea. The clock is set to 0 until the clk_dv_inc is set to the clock's period in nanoseconds. At this point the clock is set to 1 and the clk_dv is set to 0 again. A rst signal simply resets the clk_dv. As an example, the 25MHz clock has a 4ns period so the state changes when clk_dv_inc is 4. The LFSR clock within this module is a special exception as seen in the state diagram on the right. the LFSR becomes progressively faster every 3 seconds. This clock starts at 1Hz, but everytime the lfsr_3sec_counter signals that 3 seconds has elapsed, the clock's period will be decremented by a constant value such that by the end of the game the clock is at 4Hz. This clock is used to spawn moles in the game at a progressively faster rate as the player runs out of time.

debouncer.v

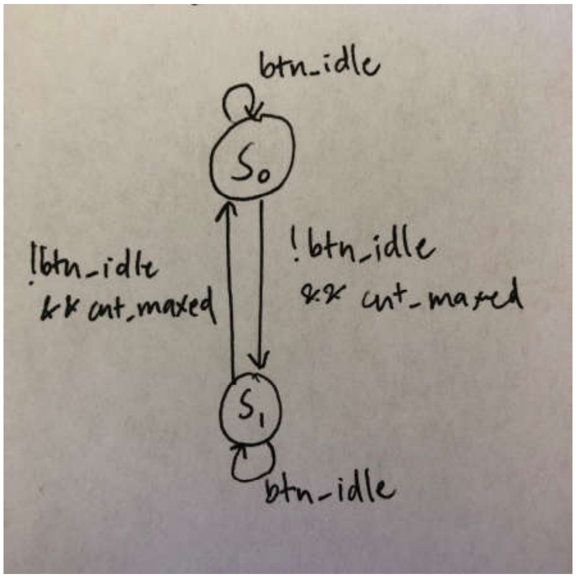


Figure 17: debouncer.v The debouncer checks for when the button is idle and when the button is idle, it stays in the current state. If the button is not idle, the debouncer will start a count to make sure that the button is pressed for at least 2.62143 ms and if this is true, it will change states. The debouncer was only used for the start and reset buttons, not the keypad buttons.

display_vga640x800.v

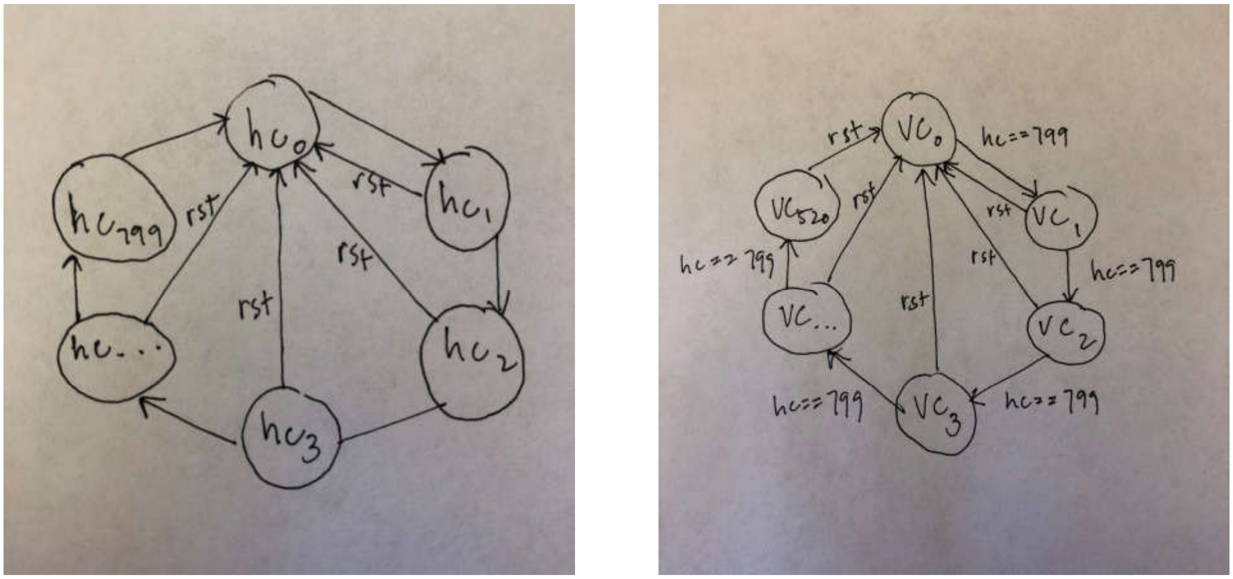


Figure 18: display_vga640x800.v The display module has a horizontal count, hc, and vertical count, vc. Both hc and vc start at 0. The value of hc is continually incremented to 799. Once hc reaches 799, vc will increment one time, so for everytime vc is incremented, hc goes one full cycle from 0 to 799. The value of vc goes up to 520. If these values reach their respective maxima or receive a clr signal, vc and hc will be set to 0. This is used to parse through every pixel of our display.

level_selector.v

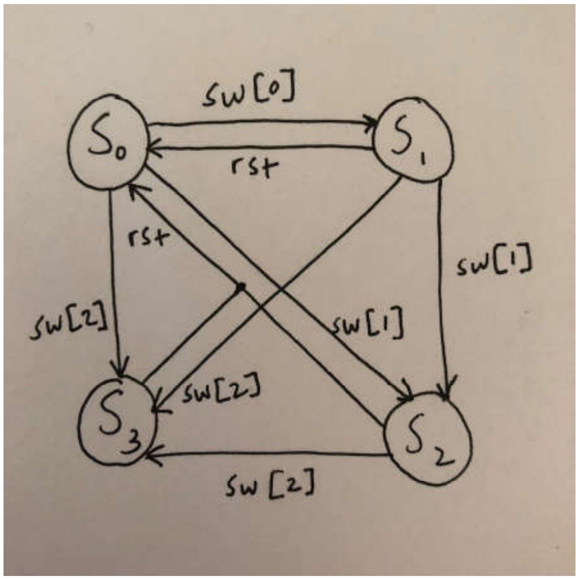


Figure 19: level_selector.v The level selector simply looked at the first three switches on the board. The level is either 1, 2, or 3 which is determined by sw[2], sw[1], or sw[0] respectively. From these 3 switches, the level selector determines the most significant bit that is set to 1 and picks that as the level. For example, if sw[2] is set to 1 then the level will be set to 3 no matter what sw[1] or sw[0] are set to. This makes it so that the player doesn't need to know binary. If all switches are set low then the level will be 0 and the game will not start. This is used for determining the level which determines the speed of the animation clock and the color of the backdrop.

mole_object.v

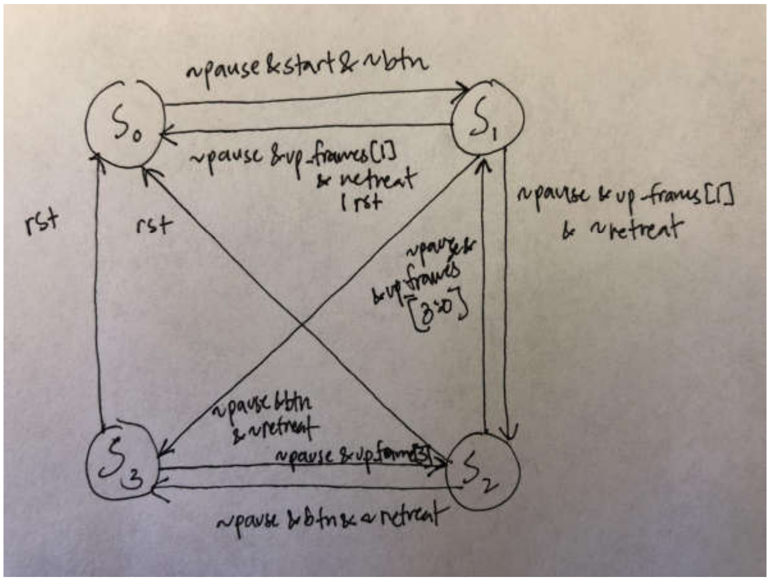


Figure 20: mole_object.v The mole can be in 4 states. State 0 is the mole in a hidden state. From this state, if the mole is given a start signal from the LFSR, and no buttons are being mashed, it will go into state 1, where the mole is deep within the hole. State 1 could either be

when the mole is coming out or retreat back into the hole after a certain number of frames. If the mole is in this state while retreating, it will go back to state 0 after a few frames. Otherwise, the mole can either register a hit by button which would go directly to state 3 or finish it's designated number of frames and proceed to state 2. State 2 can either go retreat to state 1 if it reaches its limit of up_frames, but if it registers a button press before that it will go to state 3. State 3 is the injured mole state where it will return to state 2 after its frames are completed so that the mole can begin its retreat. At any point, a mole can be set to state 0 with the reset button.

mod_7_down_cntr.v

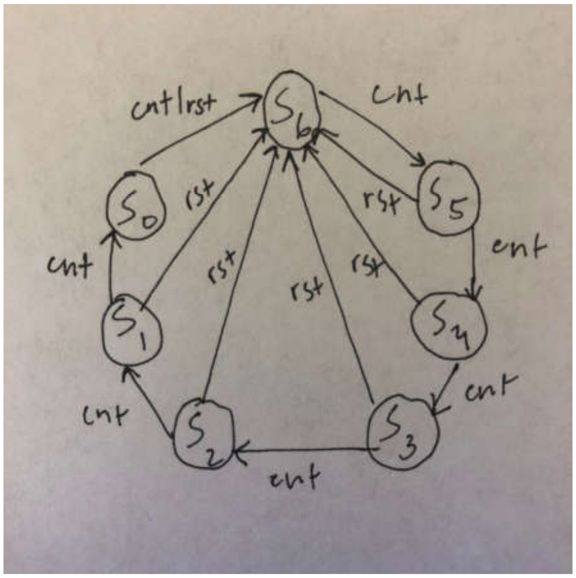


Figure 21: mod_7_down_cntr.v Given a cnt signal, the mod_7_down_cntr will count from 6 to 0 and once it hits 0 it will go back to 6 the next time it is decremented. If the reset signal is given, the counter will automatically go back to 6. This is used for the tens place of the timer within the banner of our game. We used a mod 7 counter unlike the previous project's mod 6 counter because we wanted to start the game by displaying "60" for the seconds rather than "1:00".

mod_10_updown_cntr.v

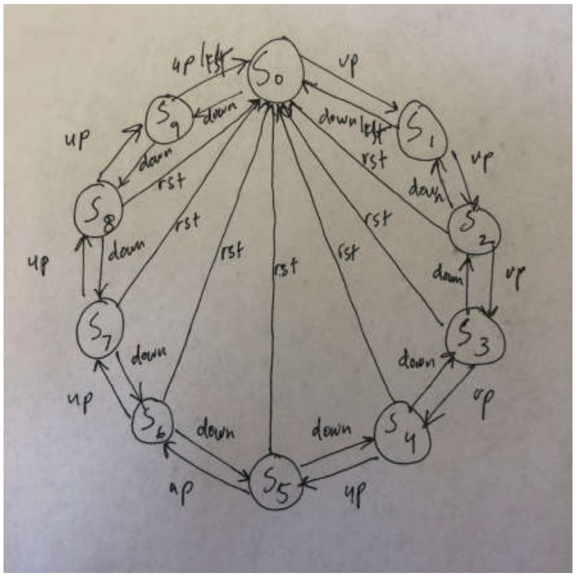


Figure 22: mod_10_updown_cntr.v The mod 10 counter works very similarly to our mod 7 counter, but rather than being given a count signal it is given either up or down. If it receives the up signal the counter will increment up to 9 where it will go back to 0 if given another up signal. If it receives the down signal the counter will decrement down to 0 where it will go back to 9 if given another down signal. If the clr bit is given at any time, the counter will go back to 0. We used this for the seconds place of our timer and the three digit points display of our game.

Simulation Tables and Waveforms

bin_to_sev_seg_TB.v

Test Case	Result	Pass/No Pass
• 0	• 7'b0111111	Pass
• 1	• 7'b0000100	Pass
• 2	• 7'b1011011	Pass
• 3	• 7'b1001111	Pass
• 4	• 7'b1100110	Pass
• 5	• 7'b1101101	Pass
• 6	• 7'b1011111	Pass
• 7	• 7'b0000111	Pass
• 8	• 7'b1111111	Pass

- 9	- 7’b1100111	Pass

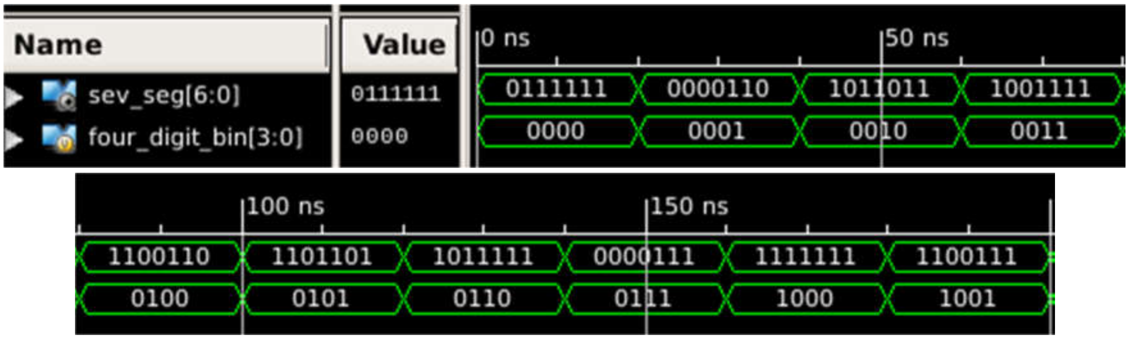


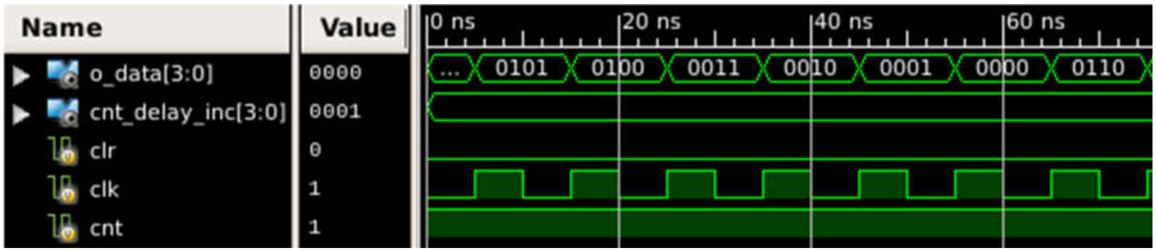
Figure 23: bin_to_sev_seg_TB.v results. The image above shows the results of execution of the test cases in the above table when testing the binary to seven segment module.

mod_7_down_cntr_TB.v

Test Case	Result	Pass/No Pass

clr = 0;
clk = 0;
cnt = 1;
	- counter counts from 6 to 0, downwards
	- wraps around to 6 from 0
		Pass

cnt is set low after being high, and then set high after begin low	- counter runs while cnt is high, maintains its values while cnt is low - can stop and resume counting with no issue	Pass
clr is set high after being low, and the low after being high	- clr will set the counter to 6 and hold that value as long as clr is high	Pass



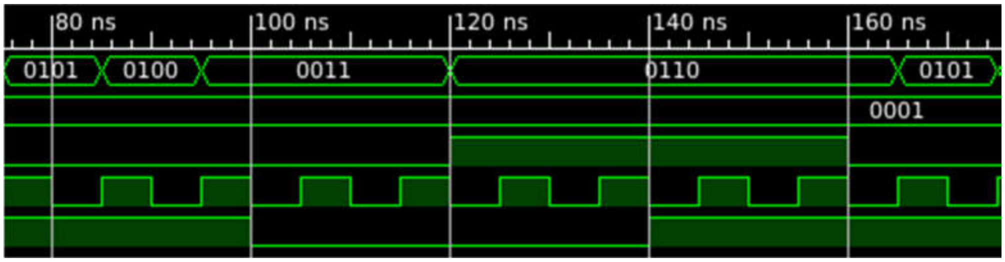


Figure 24: mod_7_down_cntr_TB.v results. The image above shows the results of the execution of the test cases in the above table when testing our modulo 7 down counter module.

mod_10_updown_cntr_TB.v

Test Case	Result	Pass/No Pass

clr = 0;
clk = 0;
cnt = 1;
up = 1;
	● counter counts from 0 to 9, upwards
	● wraps around to 0 from 9
	● terminal_cnt is high
		Pass

clr = 0;
clk = 0;
cnt = 1;
up = 0;
	● counter counts from 9 to 0, downwards
	● wraps around to 9 from 0
		Pass

cnt is set low after being high, and then set high after begin low	● counter runs while cnt is high, maintains its values while cnt is low ● can stop and resume counting with no issue	Pass
clr is set high after being low, and the low after being high this is done for up being high and low	● clr will set the counter to 0 if up = 1 and hold that value as long as clr is high ● clr will do the same thing when up = 0, except it hold a value of 9	Pass

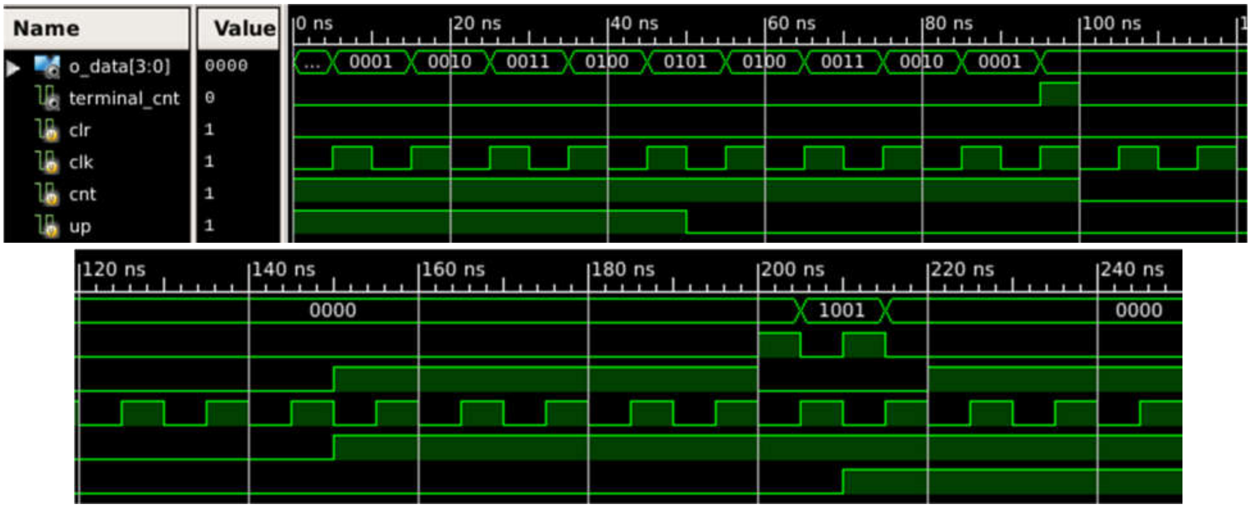


Figure 25: mod_10_cntr_TB.v results. The image above shows the results of the execution of the test cases in the above table when testing our modulo 10 updown counter module.

minute_timer_TB.v

Test Case	Result	Pass/No Pass

clr = 0;
cnt = 1;
	● The timer properly counts down from 60 to 0. The timer then proceeds from 0 to 69 but this is designed, the timer will be controlled using the clr and cnt signals	Pass

clr = 0;
cnt = 1;
The timer is run and clr is set high and then back low
	● The timer properly counts down from 60
	● It is set back to 60 when clr is written high, and holds that position for as long as clr is set
	● It begins counting down from 60 again when clr is back to low
		Pass

clr = 0;
cnt = 1;
The timer runs and cnt is set
	● The timer stops counting when cnt is low	Pass

off and on to test the timers responsiveness to a cnt signal clr is set high during a period when cnt is low to make sure the timer can be reset when paused	• The timer counts when count is high • The timer is reset to 60 whenever clr is high, regardless of cnt	

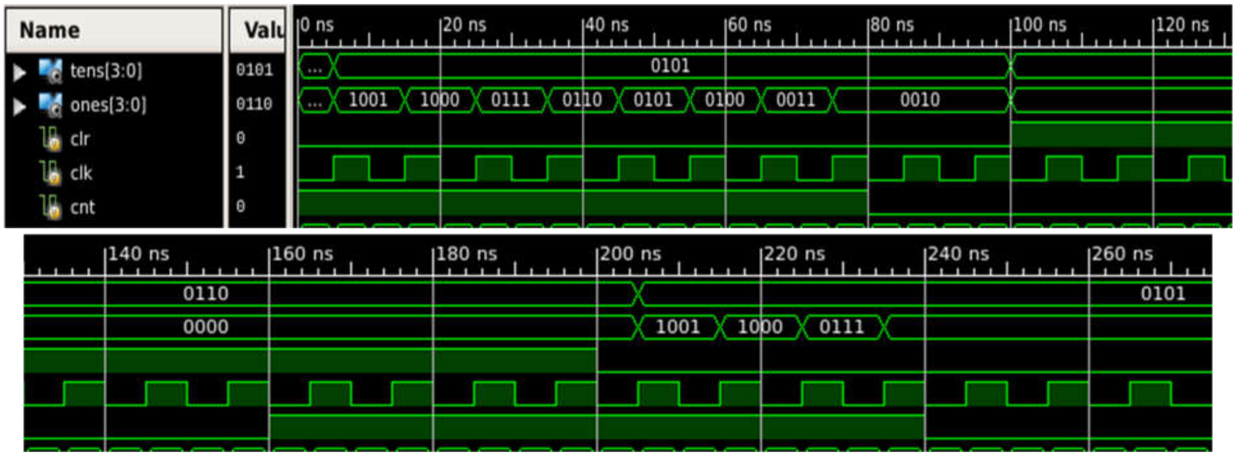


Figure 26: minute_timer_TB.v results. The image above shows the results of the represents the execution of the test cases in the above table when testing our minute timer module.

clk_divider_TB.v

Test Case	Result	Pass/No Pass
• rst = 0; • lvl = 0; / lvl = 1; (same behavior for both) The clock divider is left alone to run and the timing of the signals is observed	• the 1 Hz clock, 25 MHz clock, and the LFSR clock behave perfectly • the animation clock is 10 Hz as desired for level 0 and level 1	Pass
• rst = 0; • lvl = 0; / lvl = 2; (same behavior for both) The clock divider is left alone to run and the timing of the signals is observed	• the 1 Hz clock, 25 MHz clock, and the LFSR clock behave perfectly • the animation clock is 20 Hz as desired for level 2	

● rst = 0; ● lvl = 0; / lvl = 3; (same behavior for both) The clock divider is left alone to run and the timing of the signals is observed	● the 1 Hz clock, 25 MHz clock, and the LFSRclock behave perfectly ● the animation clock is 30 Hz as desired for level 3	

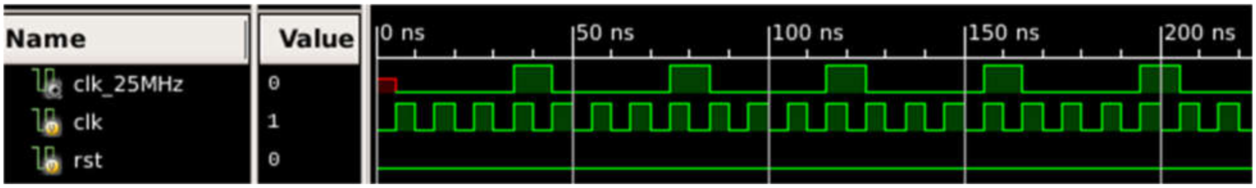


Figure 27: clk_divider_TB.v results showing 25MHz clock. The image above shows the results of the testing done on the clk_divider module, magnified to show the positive and negative edges of clk_25Mhz.

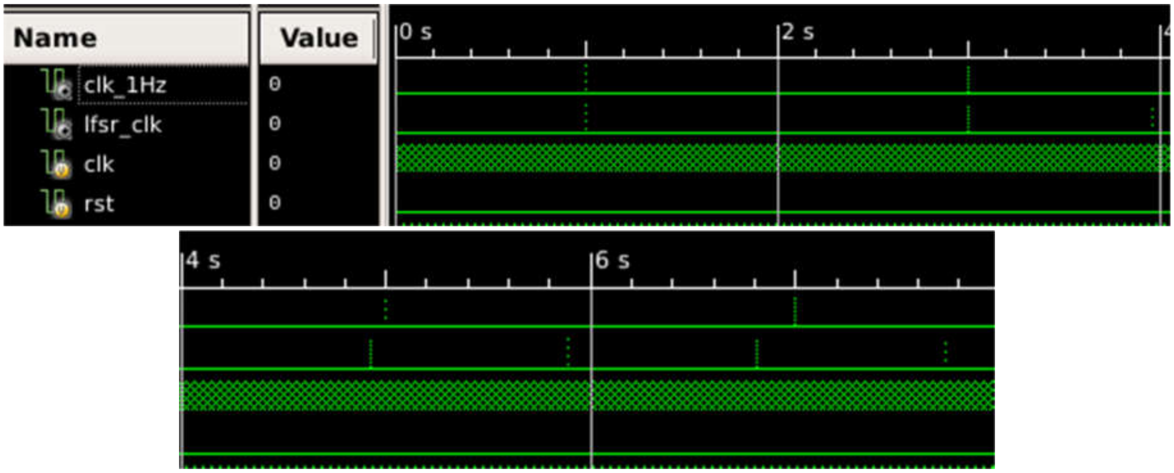


Figure 28: clk_divider_TB.v results showing LFSR clock. The image above shows the results of the testing done on the clk_divider module, focusing on the speed increase of the LFSR clk after every three seconds.

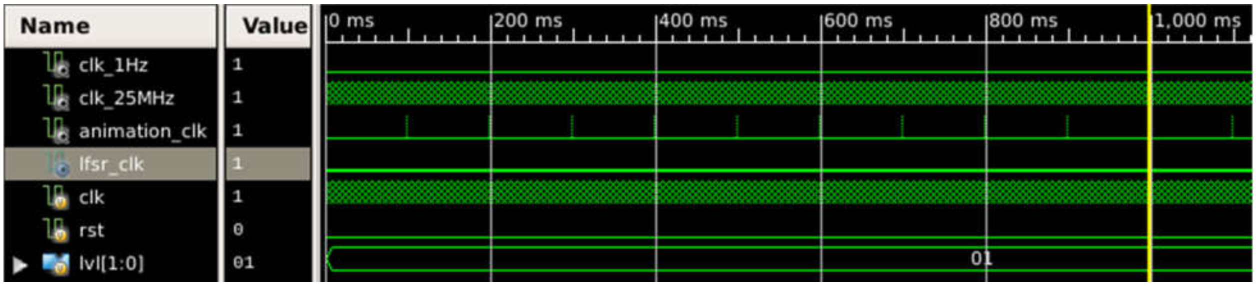


Figure 29: clk_divider_TB.v results for level 0 or 1. The image above shows the results of the testing done on the clk_divider module with a value of 0 or 1 for lvl.

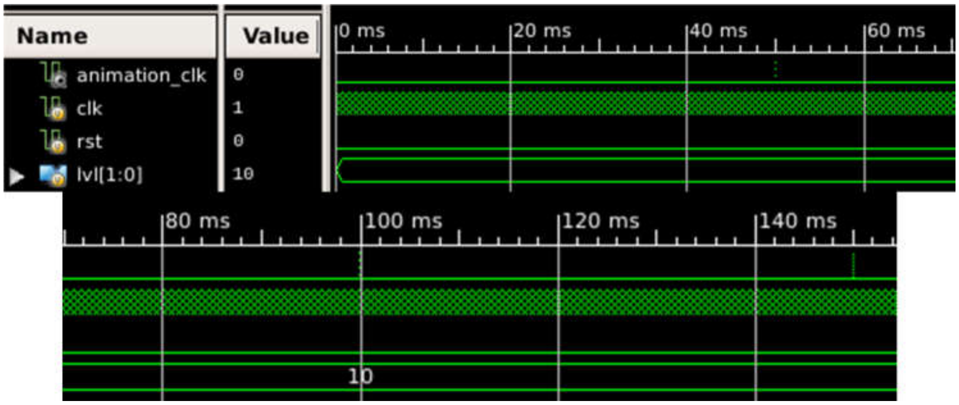


Figure 30: clk_divider_TB.v results for level 2. The image above shows the results of the testing done on the clk_divider module with a value of 2 for lvl. clk_25MHz, clk_1Hz, and lfsr_clk do not change behavior when clk_divider.v is given different values for lvl, so only the change to animation_clk is displayed in this waveform.

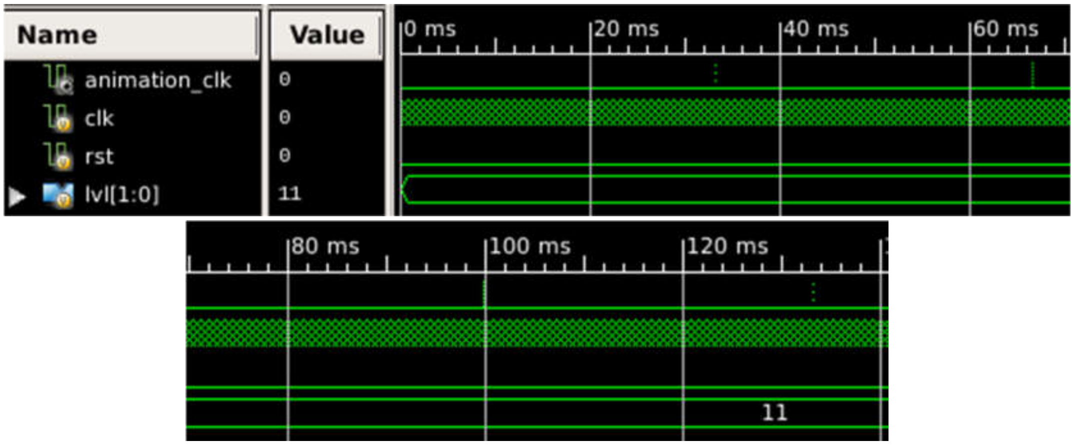


Figure 31: clk_divider_TB.v results for level 3. The image above shows the results of the testing done on the clk_divider module with a value of 2 for lvl. clk_25MHz, clk_1Hz, and lfsr_clk do not change behavior when clk_divider.v is given different values for lvl, so only the change to animation_clk is displayed in this waveform.

LFSR_10_TB.v

Test Case	Result	Pass/No Pass
rst_n starts low, then stays high so as to test the LFSR output	● random 16 bits are output at each positive clock edge	Pass

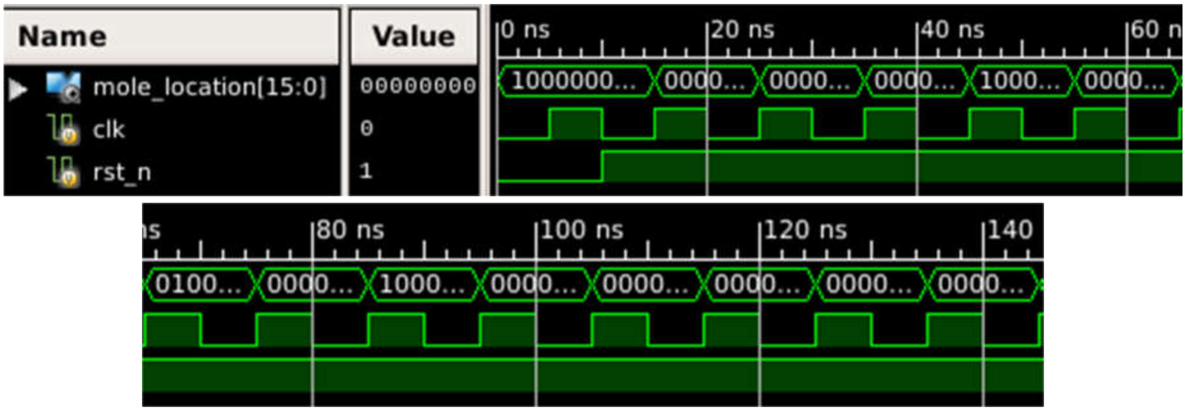


Figure 32: LFSR_10_TB.v results. The image above shows the results of the testing done on the LFSR pseudo random number generator module. The values are changing at each positive clock edge and no easily discernible pattern was present during testing.

debouncer_TB.v

A bouncing button was difficult to simulate in verilog, so most of the actual debouncing testing was done outside of testbenches and by actually working with the button itself to figure out a good counter size for debouncer.v to use to detect when a button should change state.

Test Case	Result	Pass/No Pass
btn_raw is written high for about 6ms, long enough for the debouncer code to recognize it as a push	● btn_state is set high for the same amount of time, with a significant delay of about the same length of time as the button was pressed for	Pass

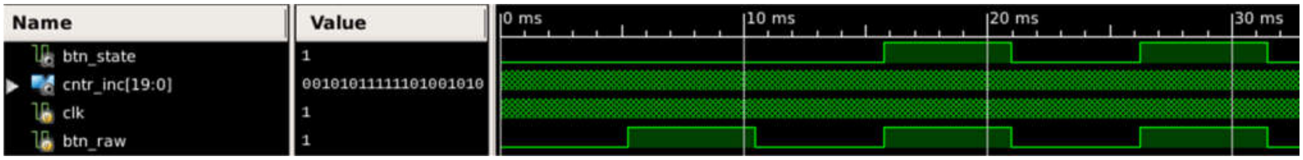


Figure 33: debouncer_TB.v results. The image above displays the results of a simple test designed to evaluate the debouncer module’s ability to detect a button press.

level_selector_TB.v

Test Case	Result	Pass/No Pass
● sw = 0;	● selects level 0	Pass

● sw = 1;	● selects level 1	Pass
● sw = 2;	● selects level 2	Pass
● sw = 3;	● selects level 2	Pass
● sw = 4;	● selects level 3	Pass
● sw = 5;	● selects level 3	Pass
● sw = 6;	● selects level 3	Pass
● sw = 7;	● selects level 3	Pass

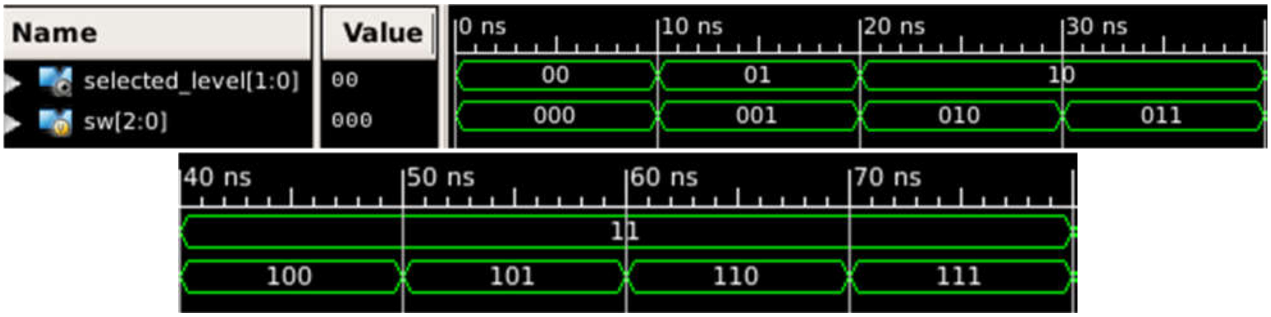
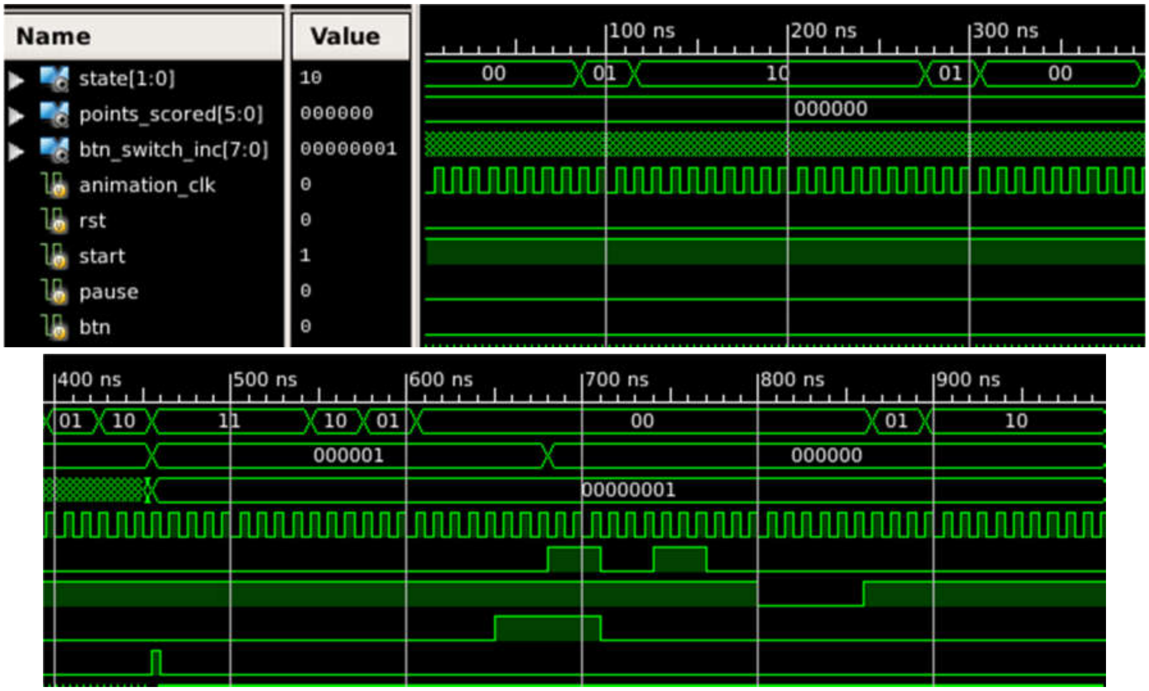


Figure 34: level_selector_TB.v results. The image above displays the results of a simple test designed to evaluate the level selector module's ability to translate the input from switches to a two bit binary level number.

mole_object_TB.v

Test Case	Result	Pass/No Pass
● rst = 0; ● start = 1; ● pause = 0; btn is pressed during a vulnerable state	● The mole advances states properly ● the mole responds to button, goes to state 3 and increments points scored ● the mole remains in a state for the correct amount of clk posedges	Pass
following above case: pause is set high while rst is low	● mole does not change state	Pass
following above case:	● state is set to 0	Pass

rst is set high while pause is high	- points scored set 0	
following above case: rst is set high while pause is low	- state is set to 0 - up_frames is set to 0 - retreat is set to 0 - points is set to 0 - these values are maintained while rst is high	Pass
following above case: rst is set low start is set low	- the mole cannot advance from state 0 (will not spawn)	Pass
start signal is sent when btn is pressed	- the mole cannot advance from state 0 (will not spawn)	Pass
btn is pressed when mole is retreating (has already been hit or the player missed their chance)	- the mole state does not change and the score does not change	Pass



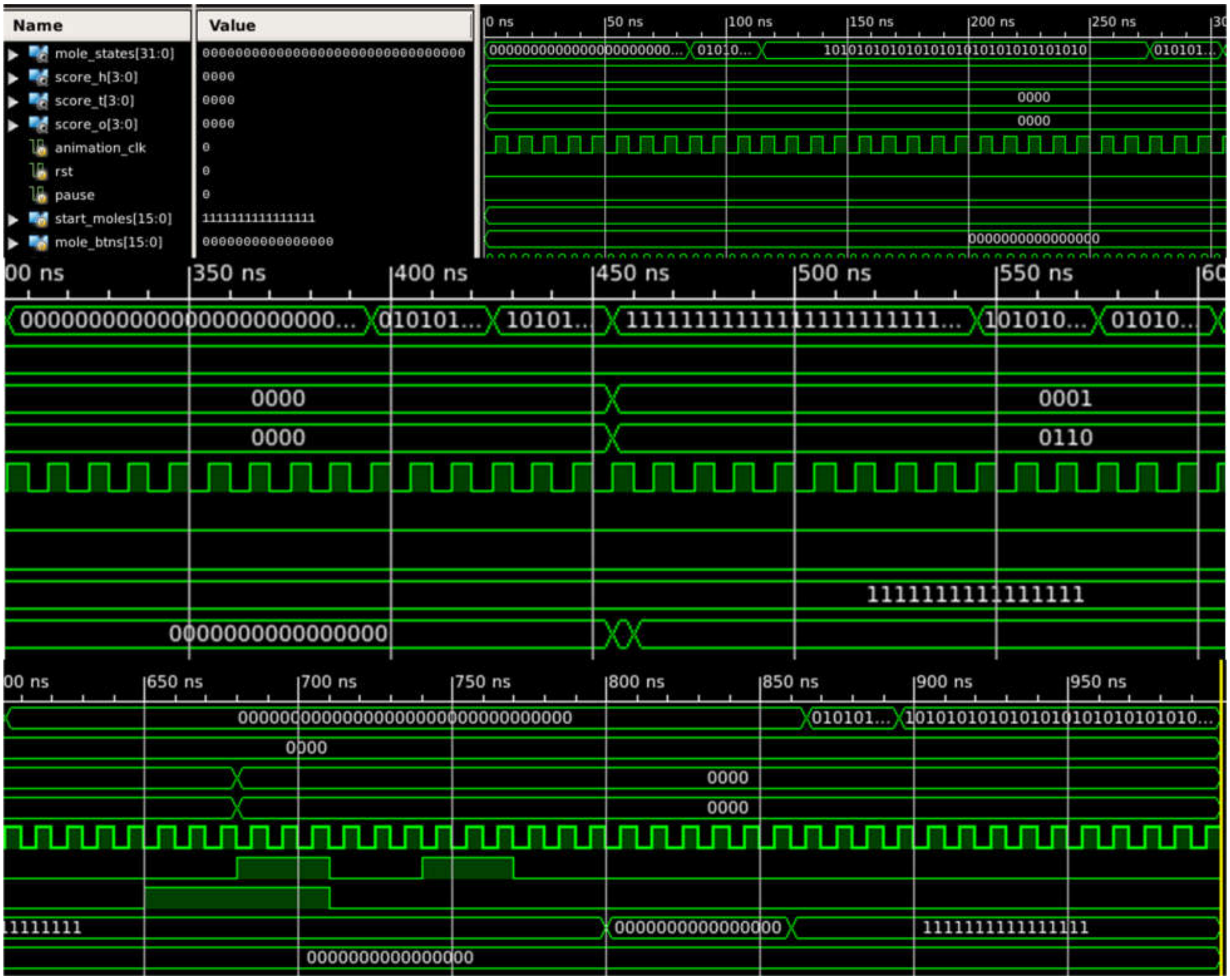
Name	Value
state[1:0]	10
points_scored[5:0]	000000
btn_switch_inc[7:0]	00000001
animation_clk	0
rst	0
start	1
pause	0
btn	0

Figure 35: mole_object_TB.v results. The image above shows the results of the testing done on the mole object module.

sixteen_moles_TB.v

The module titled sixteen_moles is extremely simple and just groups 16 separate mole modules and sums their scores. This test fixture essentially repeats the tests done in mole_object_TB.v

Test Case	Result	Pass/No Pass
● rst = 0; ● start = 16'b1111111111111111; ● pause = 0; mole_btns starts at 0 and is set to 16'b1111111111111111 during a vulnerable state. After the "press" of all mole_btns they are returned to 0.	● The moles advance states properly ● the moles responds to button, goes to state 3 and increments points scored ● the moles remain in a state for the correct amount of clk posedges	Pass
following above case: pause is set high while rst is low	● moles do not change state	Pass
following above case: rst is set high while pause is high	● states are set to 0 ● points scored set to 0	Pass
following above case: rst is set high while pause is low	● states are set to 0 ● points are set to 0 ● these values are maintained while rst is high	Pass
following above case: rst is set low start is set low	● the moles cannot advance from state 0 (will not spawn)	Pass



Name	Value
mole_states[31:0]	00000000000000000000000000000000
score_h[3:0]	0000
score_t[3:0]	0000
score_o[3:0]	0000
animation_clk	0
rst	0
pause	0
start_moles[15:0]	1111111111111111
mole_btns[15:0]	0000000000000000

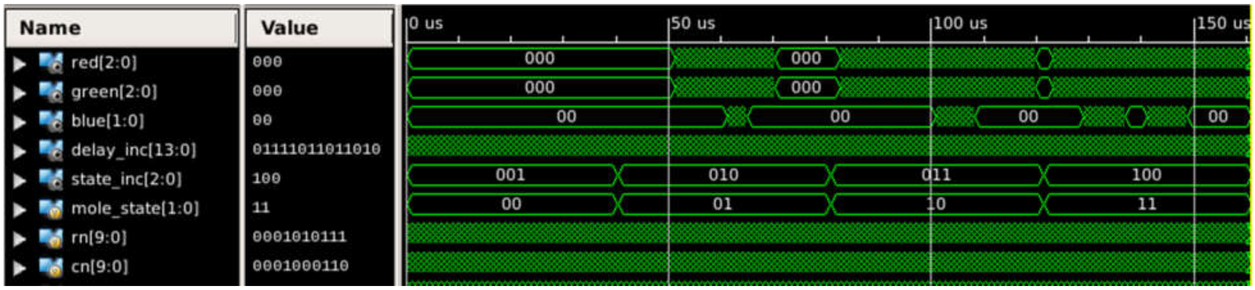
Figure 36: sixteen_moles_TB.v results. The image above shows the results of the testing done on the mole object module. While some of the numbers are cut-off in the screenshot, it is clear that they are repeating, and the input directly mirrors that of the input provided when testing with mole_object_TB.v so the output, rightly, looks the same but with sixteen times as many bits in the output (and some input).

The Xilinx simulation software dislikes inout ports so testing of the buttons was done outside of simulation by taking the number of the button pressed and outputting that to the seven segment display on the FPGA. The numpad decoder module, numpad_decode.v, was a module provided by Digilent to be used with the Digilent pmodKYPD 16 button numpad we chose to use, and numpad.v takes JA as input, feeds it to numpad_decode, and then turns numpad_decode's row and column output into a 16bit wire array. The logic was extremely simple, and testing done through physical, on-board tests was more than sufficient.

The output waveforms of vole_vga_graphic_TB.v, displayed the red, green, and blue values of each pixel and were impractically long. The testing took long hours of combing through the waveforms, looking for specific ranges of hc and vc and then making sure the output colors matched what was intended. This was practical when testing the output of mole_vga_graphic module but not so much when testing the output of the display640x480 module. Once the output of the mole_vga_graphic module was tested in testbenches, the full display output was tested in simulation through the red, green and blue output of whack_some_moles_TB because it shared the same output. Therefor, no display640x480_TB.v file was created. The display testbenches did not utilize conditional checking of the output values because the code I would have written to verify the output is a certain value when hc/cn and vc/rn would be exactly the same as the code used in the modules being tested. Therefor, the display modules, and all modules for the same reason, are just assessed by checking the waveforms produced by the testbenches.

mole_vga_graphic_TB.v

Test Case	Result	Pass/No Pass
cn and rn are incremented sequentially, so that every pixel in the 90x90 pixel box gets checked mole state is changed after every pixel has been covered	● each section of the mole is the proper color for the current state (the mole states are drawn correctly)	Pass



Name	Value
red[2:0]	000
green[2:0]	000
blue[1:0]	00
delay_inc[13:0]	01111011011010
state_inc[2:0]	100
mole_state[1:0]	11
rn[9:0]	0001010111
cn[9:0]	0001000110

Figure 37: mole_vga_graphic_TB.v results. The image above displays the change in RGB values output by the module given a change in pixel location and mole state. These waveforms were fastidiously checked to ensure the module was outputting the correct values to display the mole image for the mole_state input.

whack_some_moles_TB.v

Certain “local” wires and registers were temporarily declared as outputs for testing purposes, so as to give a better insight into the inner workings of this top module. Also, since the test fixtures dislike inouts and the debounced buttons, in the whack_some_moles module wherever a debounced button was used it was replaced with the non-debounced original signal from the board. Very intricate and complex testing was done on the board while connected to VGA display, due to timing constraints of the simulations (it takes about half an hour to simulate 12 seconds).

Test Case	Result	Pass/No Pass
Game is started normally ● sw can be set to anything ● buttons are low multiple levels are played ● btnR (level reset button) is pressed between levels ● switches select level and btnL (start button) starts the level ● digilent pmodKYPD numpad is used (upside down, by design) to hit the moles	● header displays properly ○ level section displays “L:” followed by level number that changes when new level is started ○ points section displays “P:” followed by three digit points number ○ timer section displays “T:” followed by two digit seconds timer ● grid and backgrounds display properly, change to match level themes ● moles appear at random spots, starting with appearances at 1Hz and ending with 4Hz (4 moles appearing a second) ● moles states change, and appearances change, and respond to button presses ● game freezes/pauses when timer runs out,	Pass

	still displaying the current level, final score, time of 00, and moles on the screen at the end	
● start button (btnL) is pressed when game is running and switches are changed (or unchanged)	● nothing happens	Pass
● btnR is pressed while game is running	● returns to beginning of level, paused at time displaying 60, points displaying 000 and all empty holes	Pass
reset is pressed, btnL is pressed (switches are changed or unchanged)	● behaves like the above case and once start (btnL) is pressed the selected level starts	Pass
● numpad buttons are mashed or held down in attempt to cheese the game	● moles won't spawn while their respective button is pressed, player is punished by a lower potential final score	Pass

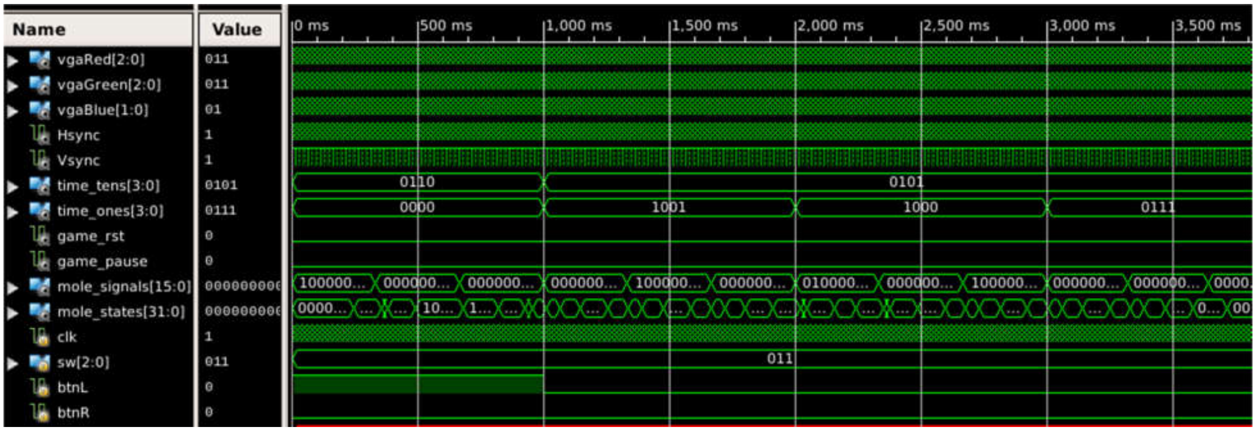


Figure 38: whack_some_moles_TB.v results. The image above displays the results of a simple test that was able to be simulated. The module is left alone and let to run, and the display values of vgaRed, vgaGreen, and vgaBlue are inspected. The mole states are correctly changing, and the logic implementing game_rst and game_pause functions properly as well.

Conclusion

Summary of the Design

Our project made use of 15 modules. The bin_to_sev_seg module was used for the header to essentially create a seven segment display similar to what was used in the stopwatch lab, and it displayed whatever number it was given, whether it be for the level, points, or time. The levels were determined through the level selector module. The points were kept track of by each individual mole storing the number of times it had been hit and summing this in the module that unified the moles into a group of sixteen. The time used the minute time module along with the mod 7 counter and mod 10 counter to determine the tens place and ones place of the time respectively. The clock divider was an essential part of this lab because it provided clocks for the timer, the speed of the animations, LFSR sampling speed, and other elements of our project. The debouncer allowed us to debounce the start and reset buttons and correctly register each press. The moles popped up in pseudo random holes through the LFSR module. The sixteen moles module simply wrapped mole_object modules within it, passing in start, pause, reset, clock, and button signals while summing their points. The numpad module registered button presses from the digilent numpad which then went to the numpad decode module which acted as a decoder for our binary values from the numpad module. The display module was used to display the game to the monitor, including a header with level number, points scored, and time left in the level. The display would update the level graphics for each level selected and used the mole vga graphics module to display different mole states in each hole. Finally, all of this was wrapped in our whack some moles module which acted as a top module and brought our game to life.

Difficulties We Encountered and Remediation

We were given a fair amount of time for this lab but we still came across many road blocks. Our biggest issue was identifying the reason behind why our moles did not display when we started the game. Initially we thought it was a problem with the LFSR not sending a start signal but later found the problem to be in the numpad module. This was extremely difficult to identify because we had to circumvent the module to get the simulations to work since the simulations always display Zs and Xs for our buttons. This simulation limitation was also present when testing our debouncer module. We also had problems with our clock speeds which hindered the game's playability but this was minor in comparison to the numpad issue, and was remedied quickly.

There were also some initial display issues, and in the end there was a minor bug in the mole logic where moles would stay in a retreating state due to a misguided route taken to try and remedy the problem that was really caused by the numpad module. Without testing on the board, it was possible to test the mole logic and the display with simulations, but we were still making assumptions about what we expected the inputs to the modules would be. As for display problems, those could be tested in simulation but not in a time efficient manner, so testing outside of class was extremely time consuming and testing and retesting various potential fixes was almost impossible unless we loaded the .bit file onto the board to determine what behavior was happening.

General Suggestions for Improving the Lab

This lab's open endedness made for a fun and rewarding lab but also made it a difficult one. This class has a very limited number of sessions for the four labs, so we can understand that we were only given seven sessions, but since this lab is worth 50% of our final grade we feel like if more time could be given for the final lab it could be beneficial. Another solution could be finding a better system for office hours. Many students found that they would need a lot of time to work on their projects outside of their designated lab sections so the last 2 weeks had an overflow of students during office hours and many students were not even able to get onto work stations because they were on a wait list. The students who were working were also kicked off to try and make room for other students. More computer stations, having extra office hours for the last two weeks, or finding a better system could help the students with these projects.

Contributions:

Robert Griffith: Vast majority of verilog code, Interface of each major module, all module diagrams except whack_some_moles.v and numpad.v, Simulation Documentation

Devan Dutta: Verilog code, Introduction, Basic Description of the Design, Modular Architecture, Interactions among the Modules

Lewis Hong: Verilog code, State Diagrams, Conclusion

Acknowledgments

We would like to thank our TA, Hongxiang, for advice during development and also for helping out at the end. His patience and dedication to his students is truly quite amazing.

We would also like to thank another TA, Fan, who opened another office hours during 10th week so that students could have more time in the lab to test their projects.

References:

- Digilent PmodKYPD numpad reference and schematics: https://store.digilentinc.com/pmod-kypd-16-button-keypad/
- LFSR pseudo random number generator references: http://www.fpga4fun.com/Counters3.html https://stackoverflow.com/questions/14497877/how-to-implement-a-pseudo-hardware-random-number-generator