Lab 4:
Minesweeper

Matthew Wong 704569674
Wenlong Xiong 204407085
Vincent Jin 604464719

CS M152A Winter 2017
Monday - Wednesday, 10 - 12

Introduction

For our final open-ended project we designed a simplified version of the popular PC game Minesweeper. In this version of the game, a 8x8 grid is populated with 10 “mines” and the rest of the squares are assigned numeric values. When the player “clicks” a square, it will display a color that represents the total number of mines in the 8 squares surrounding it. If the player “clicks” a square with a mine, he or she loses the game. Alternatively, if the player “clicks” all of the squares that do not have mines, he or she wins the game.

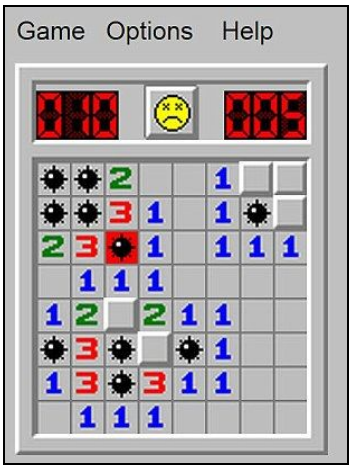

Figure 1: **Example of a Minesweeper Game.** Our version will implement a simplified version using the Nexys3 board.

We used the Nexys3 board to run the game and the board’s VGA output to display the game’s 8x8 grid on a monitor (Figure 3). In the case of a win, the entirety of the screen will turn to green, and in the case of a loss, the entirety of the screen will turn to red.

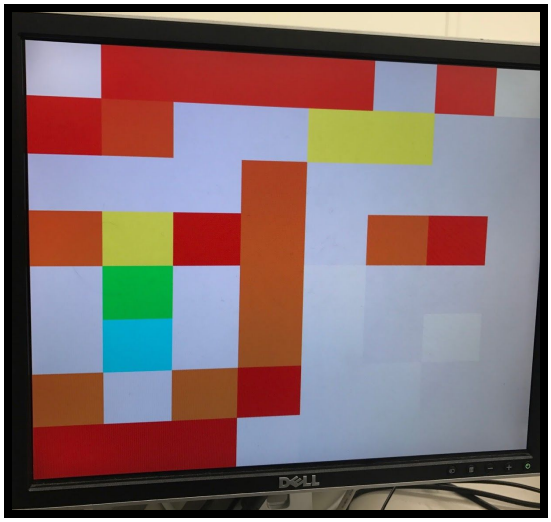

Figure 2. **Example game in progress.** Grey squares are those not yet "clicked", and colored squares are those that have been "clicked" already. the color of the square represents the number of mines surrounding it. The blinking black square (not visible here) is the current position of the cursor.

We used 5 buttons and a switch as the inputs for our game. 4 buttons are used to control the cursor position on the screen (see Figure 3), and the center button is used to "click" on the square directly under the cursor. The switch is used to reset the game state - flipping it to its on position and back to its off position again deletes the current game and generates a new random map, loading it onto the screen. In that case that you've won or lost, you must flip this reset switch to load a new random map.

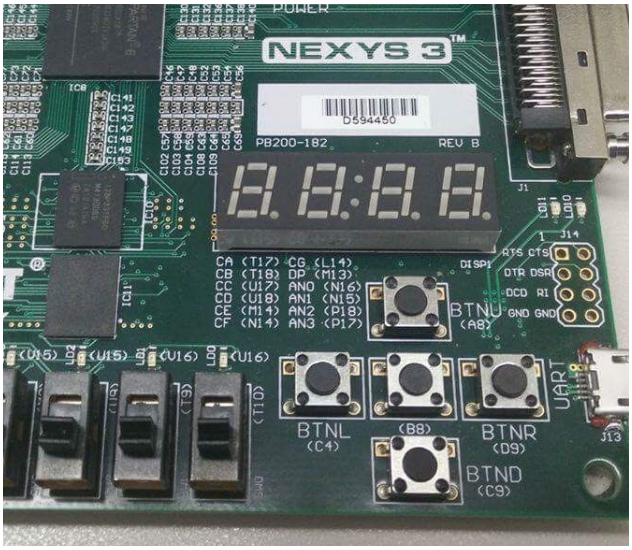

Figure 3. **5 directional and select buttons and reset switch used to control the game.** The 5 buttons on the right are the up/down/left/right and select buttons. The center button is the select, and the button to the left is the left button, button to the right is the right button, etc. The reset switch is the rightmost switch.

The maps (location of the 10 mines in the 8x8 grid) are randomly generated, based on a LFSR (linear feedback state register) that changes state every clock cycle. The LFSR outputs a pseudo-random 8-bit number, and we use the bits {7, 6, 5} as the X coordinate, and bits {4, 3, 2} as the Y coordinate of a mine in our map. Whenever the reset switch is flipped, we take 10 number outputs from the LFSR from 10 consecutive clock cycles to place 10 "mines" in the new map. While technically this means we have a limited number of "random" maps, the LFSR has enough states such that the mine's locations don't follow any visible pattern.

Design Description

The top module, called “top_game”, has inputs for the clock, five buttons, and one switch that are provided by the Nexys3 board. The five button inputs are debounced by the “controller” submodule and are used by the “map_state” submodule to determine the player’s cursor position and the clicked squares in the game. The clock has a frequency of 100 MHz, and is divided by the “clock_dividers” submodule to generate two timing signals used in the “vga640x480” submodule. The clock is also used to generate a random game map in the “map_maker” submodule. This game map is displayed to the screen by the “vga640x480” submodule and is used by the “game_state” submodule, which determines whether the game is won or lost. The switch input is used as a reset signal that can reload the game and generate a new random map.

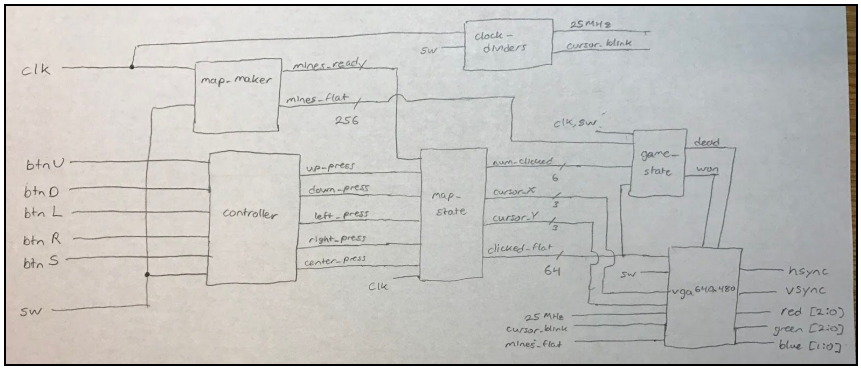

Figure 4: **High Level Design of "top_game" Module.** Nexys3 board inputs of clock, five buttons, and one switch are directed to the submodules to control the game. The game is displayed using the VGA of the Nexys3 board.

The "vga640x480" submodule is responsible for generating the VGA display output of the Nexys3 board. It uses inputs of the 'clicked_flat' and 'map_flat' to display the correct colors at the corresponding pixel values. Each space on the 8x8 grid is represented by an 80x60 pixel section of the screen. The x-y coordinates of the square correspond to a specific bit in 'clicked_flat' and 4 bits in 'map_flat'. If the square has not been clicked, the submodule will output a gray color for those pixels. Otherwise, if the square has been clicked, the submodule will output a color that corresponds to the numerical value as seen in the chart below. Also, the square that the cursor x-y coordinates point to will blink between black and the normal color according to the 4 Hz input clock. When a mine is clicked, the entire screen becomes red, telling the player the game is lost. When all non-mine squares are clicked, the entire screen becomes green, telling the player the game is won. While the reset switch is on, the screen is black since a new game is being created.

Not Clicked	0	1	2	3	4	5	6	7	8
gray	white	red	orange	yellow	green	cyan	blue	magenta	purple

Figure 5: **Numerical Values and their Associated Square Color**

The "clock_dividers" submodule contains the two clock dividers that are used by the "vga640x480" submodule. The design is the same as used in previous labs. Each divider has an internal register counter that is compared to a constant. The divider outputs a high clock signal when the counter equals the constant, creating a new frequency that is much slower than the input clock. In this submodule, the 100 MHz 'master' clock comes directly from the Nexys3 board. The 25 MHz clock uses a constant of 4 and is used to cycle through the pixel and color values of the VGA display. The 4 Hz 'cursor_blink' clock uses a constant of 2.5×10^7 and is used to create the effect of the blinking square at which the cursor is pointing.

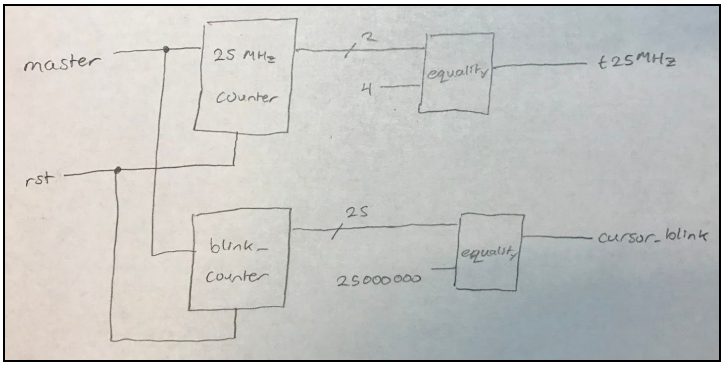

Figure 6: **High Level Design of "clock_dividers" Module.** There are two counters that each correspond to a unique clock divider. The outputs are 25 MHz and 4 Hz clock signals that are sent to the "vga640x480" submodule.

The "game_state" submodule is responsible for determining whether the game has been won or lost. It uses the flattened map values from the "map_maker" submodule and the flattened clicked values from the "map_state" submodule to determine if a mine has been clicked. If a map value is a mine and its corresponding clicked value is a 1, then the game is lost and 'dead' output becomes 1. This submodule also uses 'num_clicked' from the "map_state" submodule to determine if the game is won. If all non-mine squares have been clicked (54 squares) and the 'dead' signal is still 0, then the game is won and the 'won' signal becomes 1. The rst input from the Nexys3 switch will set 'dead' and 'won' back to 0 since the game has restarted.

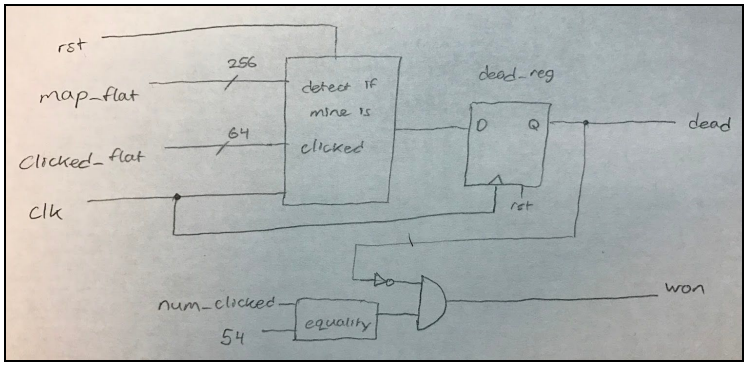

Figure 7: **High Level Design of "game_state" Module.** The 'map_flat' and 'clicked_flat' from other submodules is used to determine whether the game is won or lost. If a mine is clicked, 'dead' becomes 1. If all non-mine squares are clicked, 'won' becomes 1.

The "map_state" submodule tracks the position of the cursor and records which squares have been clicked by the player. The directional button inputs from the "controller" submodule are processed to increment or decrement the x-coordinate or y-coordinate of the cursor. These coordinates are output to the "vga640x480" submodule to display the cursor's location. The center button input is used to click a square. Clicking an unclicked square changes the corresponding register value to 1 and increments 'num_clicked'. The 2-D array of registers is flattened and output to "vga640x480" and "game_state", and 'num_clicked' is output to

“game_state”. The ‘load_new_map’ input from the “map_maker” submodule resets the 2-D array of registers to 0 and ‘num_clicked’ to 0.

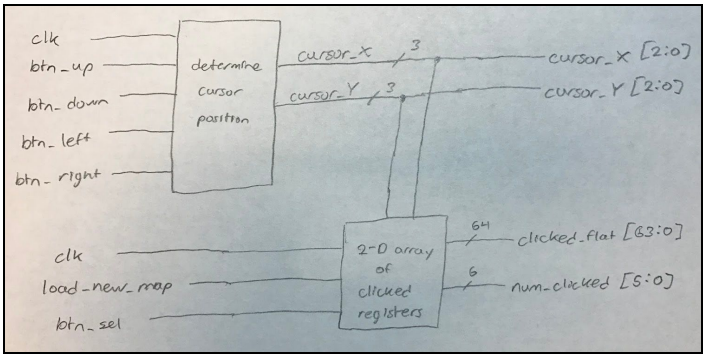

Figure 8: **High Level Design of “map_state” Module.** The directional button inputs determine the cursor’s position in the 8x8 map. The center button input is used to click the square to which the cursor points, and ‘load_new_map’ resets the submodule to its beginning state at a new game.

The “controller” submodule is an organizational submodule that channels the raw button inputs and switch input from the Nexys3 into “btn_debouncer” submodules. The controller collects the debouncers’ outputs and sends them to the next submodule, which is the “map_state” submodule.

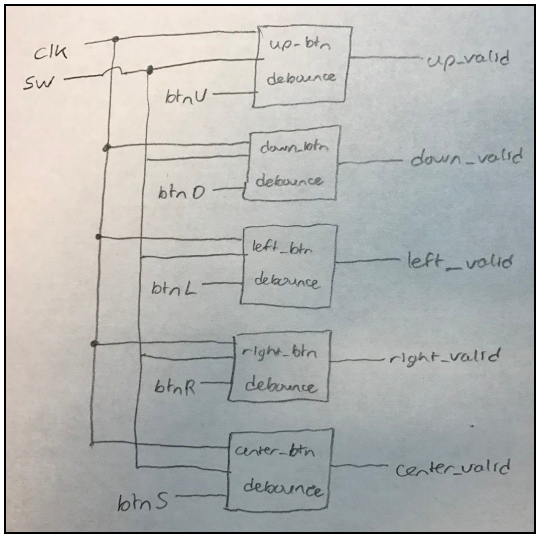

Figure 9: **High Level Design of “controller” Module.** The raw button inputs from “top_game” are sent to their own debouncers. The outputs from the debouncers are captured and are sent to the “map_state” submodule.

The “btn_debouncer” submodule is the same as used in previous labs. It is used to filter noise from buttons whenever they are pressed. To achieve this, the debouncer contains its own clock divider that creates a 763 Hz timing signal. The debouncer samples the button input at this slower frequency to determine a valid button push. When there is a valid button push, the debouncer will briefly output a high signal that will be collected by the controller submodule.

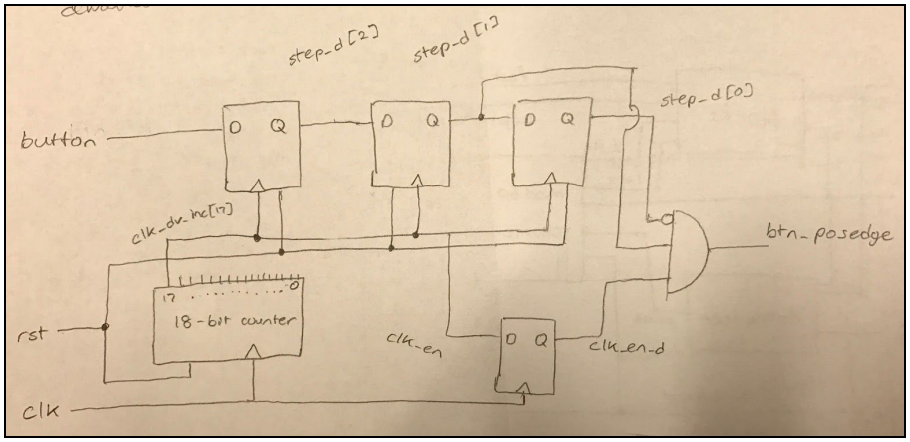

Figure 10: **High Level Design of "btn_debouncer" Module.** The input from the button is sampled at a frequency of 763 Hz. The values are stored in stepping registers that are used to determine valid button presses.

The "map_maker" submodule creates a new game map using the pseudo-random 8-bit numbers generated by the "rand_8b" submodule. The 8-bit number is split to determine the location of a new mine in the 8x8 map. Bits 7, 6, and 5 represent the x-coordinate and bits 4, 3, and 2 represent the y-coordinate of a new mine. When the mine is successfully placed (it is not placed on an already existing mine), the numerical values of the squares surrounding the mine are incremented. When 10 iterations of mine placement have been completed, the 'ready' output becomes 1, signaling to the "map_state" submodule that a new map has been created.

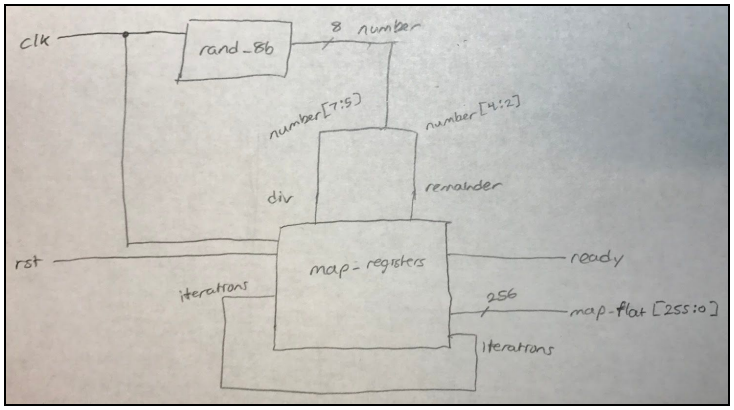

Figure 11: **High Level Design of "map_maker" Module.** The bits of the pseudo-random number from "rand_8b" are used to determine the location of a new mine in the map. When 10 mines have been placed, the map is sent to "vga640x480" and "game_state" to be used for a new game.

The "rand_8b" submodule generates and outputs a pseudo-random 8-bit number at every clock cycle. The numbers are based off an initial seed of 8'b11000101, but this seed can be easily changed in the Verilog code. To obtain the next number, the 7th, 5th, 4th, and 3rd bits

are XOR'ed, and their result is inverted. This new bit, called 'feedback', is shifted into the register as the LSB, while the previous number is left-shifted a single bit.

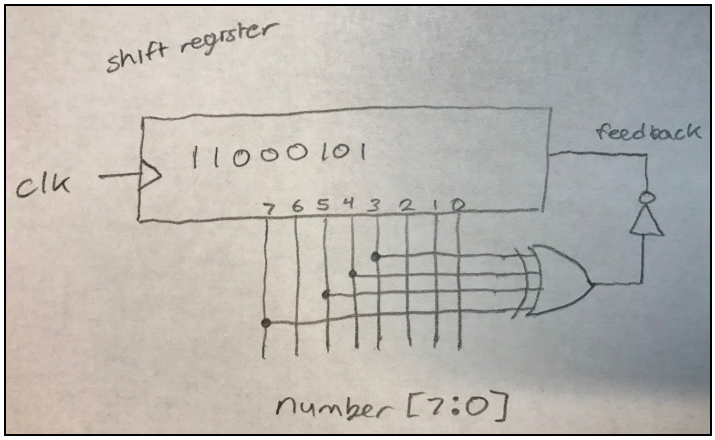

Figure 12: **High Level Design of "rand_8b" Module.** The shift register is initialized to 8'b11000101 and outputs the random 8-bit number. The next random number is determined by the feedback of some of the previous bits.

Simulation Documentation

rand_8b module:

Test Case	Result	Pass/No Pass
#5 clk = ~clk	Every positive edge of the clock should output a random 8 bit number	Passed

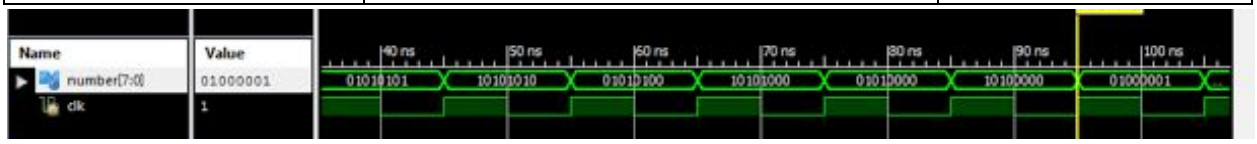

Figure 13: **Waveform of "rand_8b" Module.** At every positive edge of the clock, the 8-bit output changes to a new random number that is generated by XOR gates and shifting.

map_maker module:

Test Case	Result	Pass/No Pass
rst = 1	'map_flat' should have all values set to 0 and the 'ready' signal should be 0	Passed
rst = 0 (Normal map generation)	'map_flat' should be the flattened representation of an 8x8 grid of 4-bit values. After 10 iterations of randomly placing mines, 'ready' will equal 1.	Passed

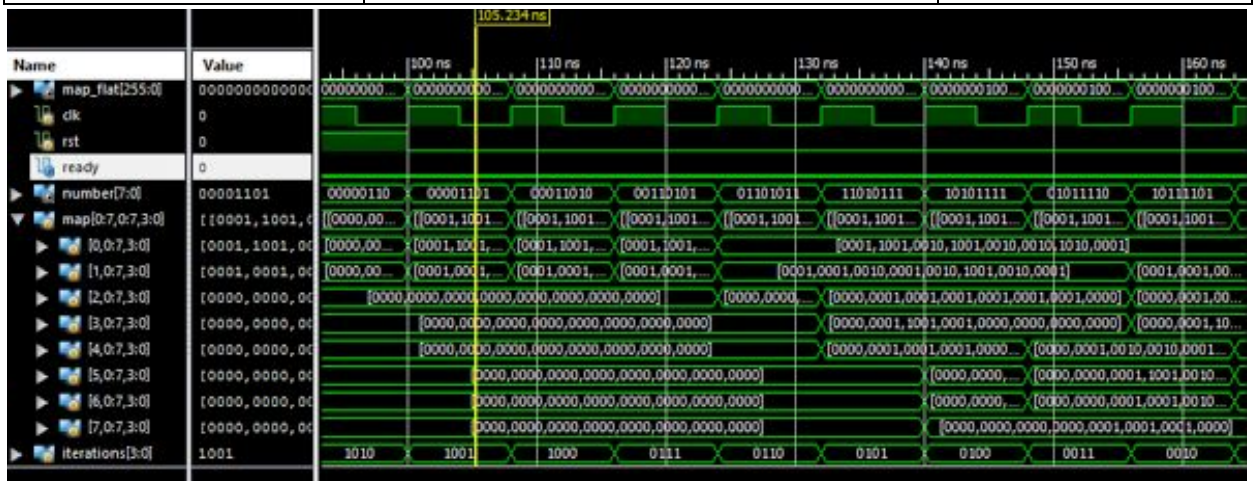

Figure 14: **Waveform of "map_maker" Module Generating Random Mines.** The module performs 10 iterations of placing mines in the map. The location is randomized by the "rand_8b" module, and the squares around each mine are incremented in number value. The waveform shows the values of the map changing with each mine placed.

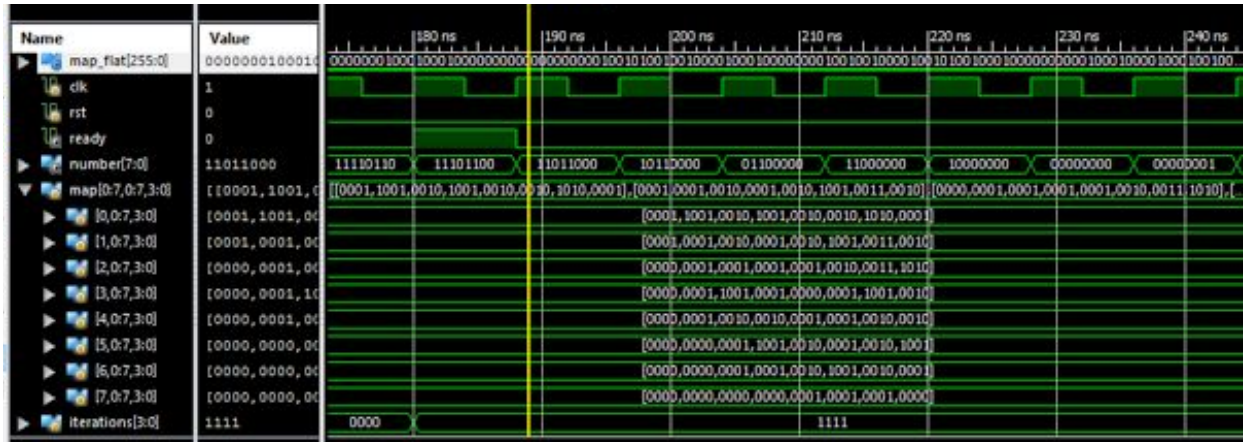

Figure 15: **Waveform of "map_maker" Module at Completion of Map.** After placing all 10 mines, the 'ready' signal is set high and the 3-D array of registers is flattened into a 256-wire array to be sent to the next module.

btn_debouncer module:

Test Case	Result	Pass/No Pass
Push FPGA button connected to debouncer	btn_debouncer module should send a signal to the controller module, which should be seen when the cursor moves (up, down, left, right) or a square is selected (center)	Passed

controller module:

Test Case	Result	Pass/No Pass
button inputs from FPGA are organized and sent to debouncers, which output high signals when buttons are pressed	a button press from FPGA is sent to controller module, which channels the button input to a debouncer. The output should be noticed at cursor movement (up, down, left, right) or square selection (center)	Passed

map_state module:

Test Case	Result	Pass/No Pass
btn_up = 1	Cursor moves up one square, so 'cursor_pos_X' decrements and 'clicked_flat' stays the same	Passed
btn_down = 1	Cursor moves down one square, so 'cursor_pos_X' increments and 'clicked_flat' stays the same	Passed

btn_left = 1	Cursor moves left one square, so ‘cursor_pos_Y’ decrements and ‘clicked_flat’ stays the same	Passed
btn_right = 1	Cursor moves right one square, so ‘cursor_pos_Y’ increments and ‘clicked_flat’ stays the same	Passed
btn_sel = 1	The square that the cursor is pointing to has its clicked value set to 1. If the square was previously unclicked, ‘num_clicked’ is incremented	Passed
Two or more buttons = 1	In the unlikely event that more than one button signals are 1 on the same clock cycle, only process one of the buttons.	Passed
load_new_map = 1	All squares are set to unclicked, and ‘num_clicked’ is set to 0	Passed

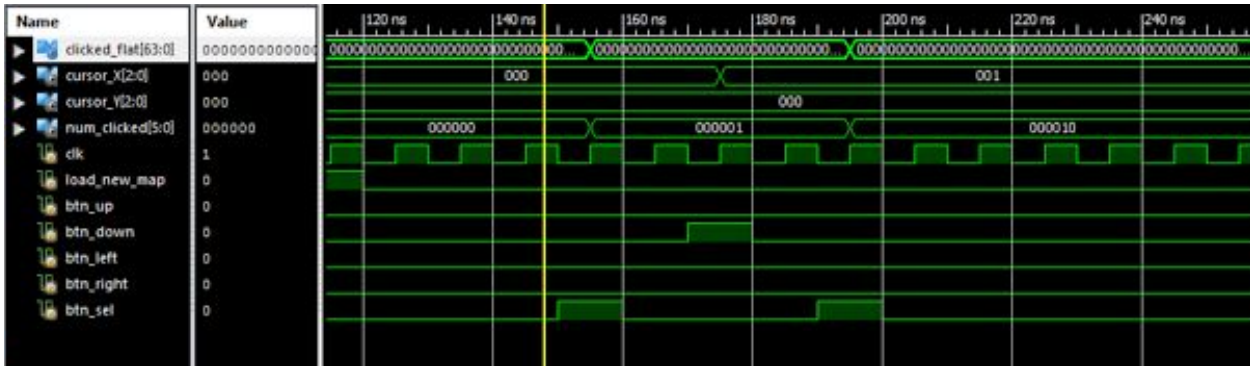

Figure 16: **Waveform of “map_state” Module Receiving Single Button Inputs.** When ‘load_new_map’ is 1, the ‘clicked_flat’ and ‘num_clicked’ outputs are reset to 0. The select button is pressed, resulting in the update to the ‘clicked_flat’ and ‘num_clicked’ output. Then the down button is pressed, moving the cursor one square. Then the select button is pressed again, updating the ‘clicked_flat’ output to show 2 clicked squares.

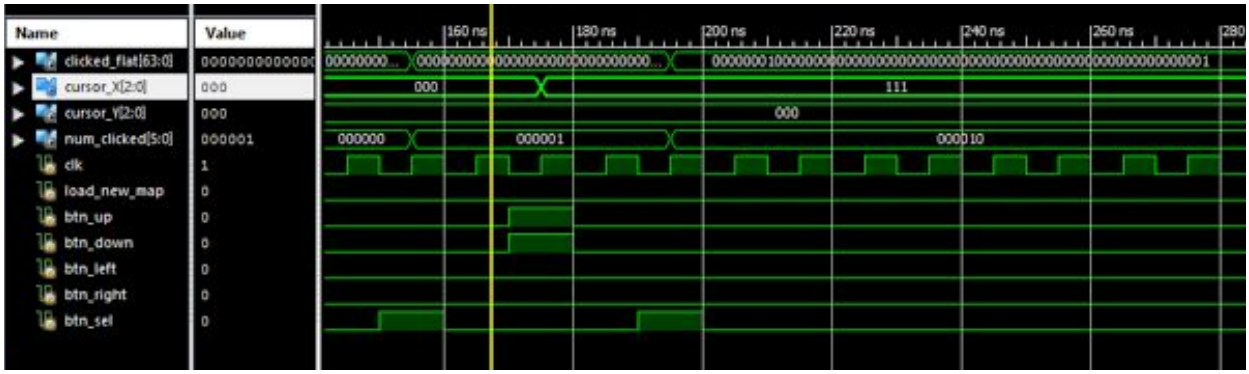

Figure 17: **Waveform of “game_state” Module when Two Button Inputs Are High.** Both the up and down buttons are pressed at the same time, resulting in the two signals equaling 1 on the same clock cycle. Only the up button is processed, so the x-coordinate of the cursor is decremented.

game_state module:

Test Case	Result	Pass/No Pass
rst = 1	reset the ‘won’ and ‘dead’ signals to 0 since the game is being reset	Passed
map_flat[x][y] >= 9 and clicked_flat[x][y] == 1, meaning a mine has been clicked	set ‘dead’ to 0, which should be noticed by the screen displaying all red	Passed
num_clicked = 54 and no mines have been clicked	If player has not died, set ‘won’ to 0, because player has clicked all 54 squares that are not mines. The screen should display all green	Passed

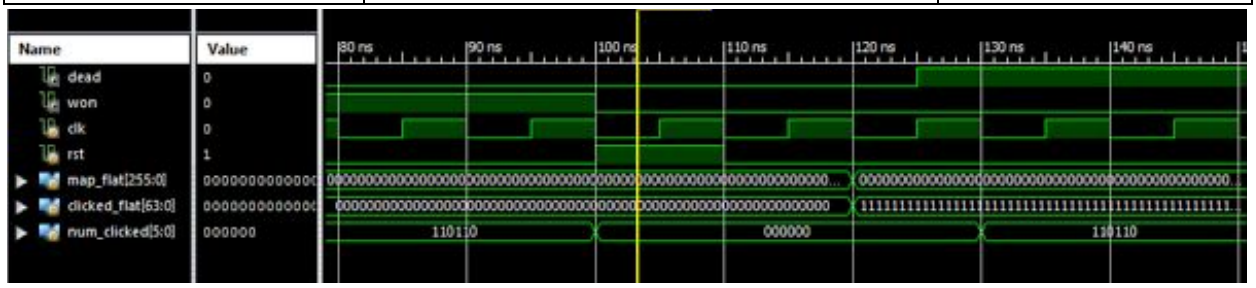

Figure 18: **Waveform of “game_state” Module.** When ‘num_clicked’ equals 54 squares, the module signals that the game is won and no more squares can be safely clicked. Then, ‘rst’ is set high, which resets the ‘dead’ and ‘won’ signals to 0. Then ‘map_flat’ is updated to contain a mine and ‘clicked_flat’ is updated to have all squares clicked, which results in the high ‘dead’ signal.

clock_dividers module:

Test Case	Result	Pass/No Pass
rst = 1	Clock dividers do not count and remain at a value of 0	Passed
rst = 0	Clock dividers count normally	Passed

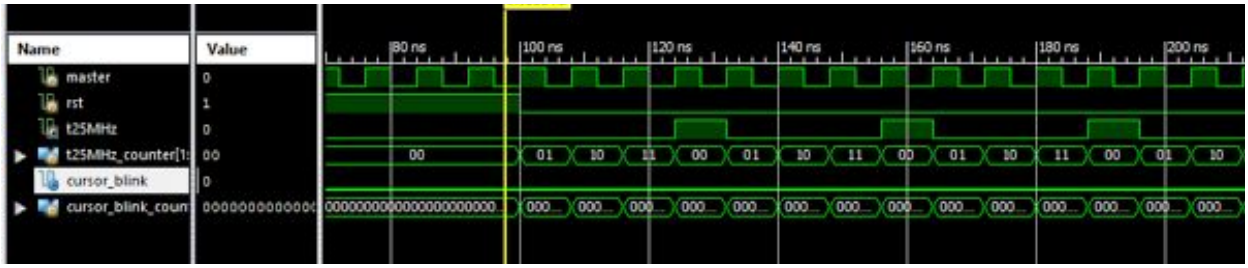

Figure 19: **Waveform of “clock_dividers” Module.** When ‘rst’ is set to 1, the counters corresponding to the clock dividers are set to 0. The 25 MHz clock divider for VGA is set to high every 4 ticks of the 100 MHz ‘master’ clock. The 4 Hz clock divider used for the blinking effect of the cursor counts much higher and is not triggered in the waveform.

vga640x480 module:

Test Case	Result	Pass/No Pass
Debug: turn off the ‘won’ and ‘dead’ signals and click all squares to show all 10 mines	Since all the numerical color values are turned off, clicking all the squares should result in just the mines being displayed as black squares in the grid	Passed
Beginning of game: no squares clicked	The screen should have all squares gray and should have the blinking cursor.	Passed
Middle of game: some squares clicked	The screen should show the unclicked squares in gray and should show the numerical color values of the clicked squares. The blinking cursor should be present.	Passed
End of game: all squares without mines have been clicked and the game is won	The screen should display a green screen when all possible squares have been clicked. (when the ‘won’ signal is equal to 1)	Passed
End of game: mine has been clicked and the game is lost	The screen should display a red screen when a mine is clicked. (when the ‘dead’ signal is equal to 1)	Passed
Reset: reset switch is flipped on to create a new game	The screen should display a black screen while the program is creating a new game map and resetting register values. After the reset switch is turned off, the display should go back to all squares being gray.	Mostly Passed - the reset switch occasionally takes 2 flips on/off to reset the screen and game

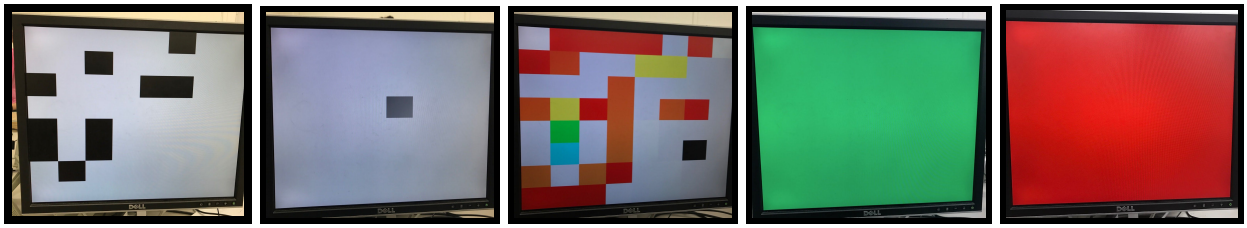

Figure 20: **Game Displays.** The left screen is captured from debugging, in which all 10 mines are denoted as black squares. The second screen shows the beginning of the game when all of the squares are unclicked and gray with the blinking cursor. The second screen is mid-game, when some of the squares are clicked and their numerical color values are visible with the blinking cursor. The third all-green screen is seen when the game is won. The last all-red screen is seen when a mine is clicked and the game is lost.

Conclusion

The simplified version of Minesweeper that we have implemented uses the Nexys3 and its VGA output to control and display the game to a monitor. There are six main submodules within an encapsulating top module. The “map_maker” submodule uses a pseudo-random number generator submodule to create a randomized 8x8 grid of mines and numeric values. The “controller” submodule debounces all of the raw button inputs from the board and outputs the resulting button signals to the “map_state” module. The “map_state” submodule records the squares in the 8x8 grid that the player has clicked and the tracks the x-y coordinates of the player’s game cursor. The “game_state” submodule uses the map from “map_maker” and the clicked spaces from “map_state” to determine whether the player has won or lost the game. The “clock_dividers” submodule creates timing signals for the “vga640x480” submodule that displays the game grid and endgame screens on the monitor.

One of the difficulties we encountered was creating a module that pseudo-randomly generated a new map for each new game. During our first drafts of the “map_maker” submodule, we used ($random)%8 to randomly generate numbers that could be manipulated into the x-y coordinates for mines. This approach worked for simulation, but $random does not work in implementation. Thus, we had to create the “rand_8b” submodule that creates pseudo-random 8-bit numbers based on an initial 8-bit seed number. Since this submodule could only return a single 8-bit number per clock cycle, we had to create an “artificial for-loop” whose iterations place a single mine in the map every clock cycle. When the 10 iterations completed, the “map_maker” submodule outputs a ‘ready’ signal to the other modules that the new map is complete.

Another problem faced was being introduced to programming the VGA display. The VGA display has several “rules” that we were not aware of when first writing the Verilog code. For example, the front and back porches of the display are required to have their color as black to allow the rest of the display to function properly and display the correct colors.

One improvement we would definitely make to our Minesweeper program would be to implement the VGA to display numeric characters instead of the solid colors that represent numbers. However, due to the lab’s quick deadline, we focused on the functional aspects of the program, for which the solid colors were satisfactory for the game’s playability.