CSM152A Lab 4: Creative Lab - Sudoku

Jerin Tomy, Ying Bin Wu

I. Introduction

Overview

In this lab, we decided to create the game of Sudoku using the FPGA board. The VGA output was used to display our current state of the board. Buttons on the Nexys3 FPGA board were utilized to allow the player to navigate and change the state of the Sudoku grid. Like lab 3, we also used the seven segment display to show the amount of time left for the game when the game is still running or the current and high score when the player has either won or lost. Additionally,a switch was used that allowed players to reset the state of the game while maintaining the current high score. These features allowed us to create a new game in the case that the player wants to start over. If this happens, the current high score still persists through all the games played so far.

Game Rules

The grid in Sudoku is comprised of 81 individual tiles, first partitioned into nine 3x3 blocks, and then these 3x3 blocks are arranged into a larger 9x9 grid, as shown in the figure below:

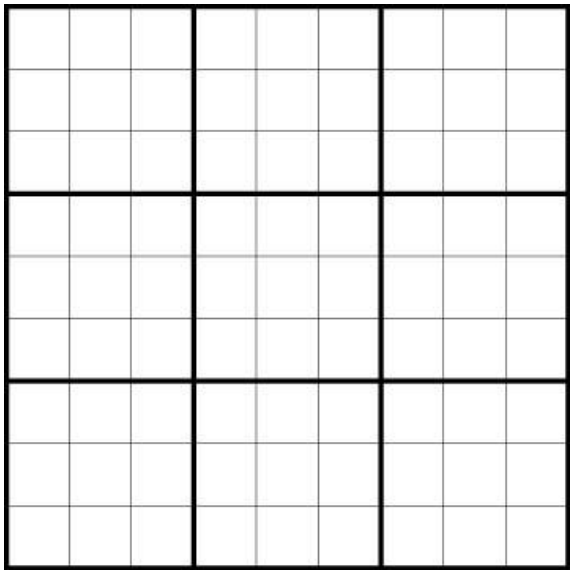


Figure 1: An empty Sudoku board showing the individual subgrids within the larger grid

There are three subsets of the total grid that the player must consider; columns, rows, and the aforementioned 3 x 3 block. The purpose of the game is to fill the grid so that each individual tile in a row, column, and block have a unique color(except black or white since it represents empty

or current position respectively). Prior to the game starting, some of the tiles in the grid will already have colors in them, in such a way that there is only one solution to the game.
Players have 900 seconds to properly fill the grid. If they do so successfully, the seven segment display will be updated a score dependent on the finishing time; otherwise the player is notified that they ran out of time and will be restricted in making further moves.

The best score will be displayed, so that the player has an objective to beat.

If the player feels the need to start over, he/she can flip a switch on the FPGA to reset the grid , and the time will also be reset.

II. Design Description

Our design was separated into the following modules:

top_module

To implement this design, we first divided the 100 Mhz master clock from the Nexys3 board and from it created a clock for the seven segment display, a clock for updating the pixels, and a clock for decrementing the timer. We then had to debounce the buttons used for navigation on the Nexys3 FPGA board and check for a rising edge for each button. This indicates that the button was pressed at some instance in time. To allow users to move and navigate the Sudoku grid and change it's state, we initialized an array with preset values in it to represent the initial state of the grid. We then update our position on the board or the color at a given position when we have detected that the user pressed one of the buttons on the FPGA board. Based on the numbers in our array and our current position, we display different colors on our VGA output. We had a total of 11 possible colors for any given square on the grid. A white represents the position of the grid we are currently on. A black box represents an empty box on the grid. Red, orange, yellow, green, cyan, blue, purple, burgundy, and pink represent the different possibilities that each box can have. Afterwards, we set up the timer for the seven segment display indicating the time remaining to solve the puzzle. In order to check if the player has won, a comparison was done between the current state of the board and the winning board. To display the time, the numbers displayed on the seven segment were rotated with a high frequency so that the human eye cannot detect the changes. When the solution to the puzzle has been found, a signal is sent to the rest of the circuit to stop play and flash between the high score and the current score on the seven segment display.

clockdiv

We used the 100 MHz master clock from the Nexys3 board to create 3 additional clocks: the clock used for driving the seven segment display, the clock used for driving the pixel updates, and the clock used to decrement the timer. The seven segment, pixel, and timer clock ran at 381

Hz, 25 MHz, and 1 Hz respectively. To create these clocks, a counter for each clock was kept that increments on each cycle of the master clock. The corresponding clock values flipped whenever the value for its designated counter counter was equal to the number of cycles for it's given frequency.

edge_debouncer

In our implementation of Sudoku, the buttons on the FPGA board had to only register one "toggle" per click. In order to do this, we needed to detect the rising edges of the button clicks. However, before we did the detection of the rising edge, we needed to first debouncer the buttons to eliminate the noise when a button is pressed. To do this, a slower clock was used for sampling so the output doesn't pick up on the noise between the button being pressed and the button being off. After the debouncing was done, the rising edge detection was done by keeping a record of the button history. By keeping track of a button history register, which represents the registered value for the button in the last 3 cycles, we can do an explicit check to see if one value is low and the preceding value is high. In this case, we set out output to be high for that small instance in time. This creates a signal that is high for a small window of time after the button was pressed, effectively representing the moment in time in which the button was pressed.

board_logic

In order to represent the change in state of our board, the old state of the board must be known as well as the buttons that were clicked. Additionally, win_flag and lose_flag, indicating whether we won or lost, must be known. In the board_logic module, the grid was first initialized to its starting state and its starting position was set to the upper left hand corner of the grid. If win_flag or lose_flag is set, we do nothing. This allows the user to see their results after the game has finished, but restrict them from changing the state of the board. Whenever the "select", the button between all of the directional buttons on the FPGA board, is pressed, the value on the board is changed to the next value up as long as our current tile isn't part of the initial state, otherwise it does nothing. If a directional button is pressed, the selected tile is changed based on the direction that was just pressed. This can change the value of the selected tile to be the one directly on top/bottom/left/right of it, but if it is on the edge, it loops over to the other end of the grid. The value of select is outputted in the case that other modules want to be able to see the currently selected tile. If clr is ever triggered, we reset the state of our board to be its initial state and reset the currently selected tile to be the top left tile.

vga640x480

For the player to be able to visualize the current state of the board, the VGA output was utilized. In order to do this, the grid and current position needed to be a parameter into the module. Additionally, we took in clr and dclk as parameters. Clr outputs a black screen indicating that we want to reset our board to its initial state if set to high. Dclk is the pixel clock, serving as the rate

in which we want our pixels to update. Each individual tile is associated with a color based on its current value on the board and whether or not it is the tile corresponding to our current position. If the tile has no value in it yet, we give it the color black. If it is associated with a value, we output the color corresponding to that value. The colors we chose are as follows: Red, orange, yellow, green, cyan, blue, purple, burgundy, and pink. These colors corresponded with the values 1-9 respectively. In order to indicate our current position on the board, we flashed the tile corresponding to it with its current color and white. To output these colors, we had to look up the RGB values for each individual color. Since the color values we looked up correspond with a 24 bit value, each value had to be compressed into an 8 bit RGB value to feed the output of the VGA. This was done by running the RGB values through an algorithm we found online.

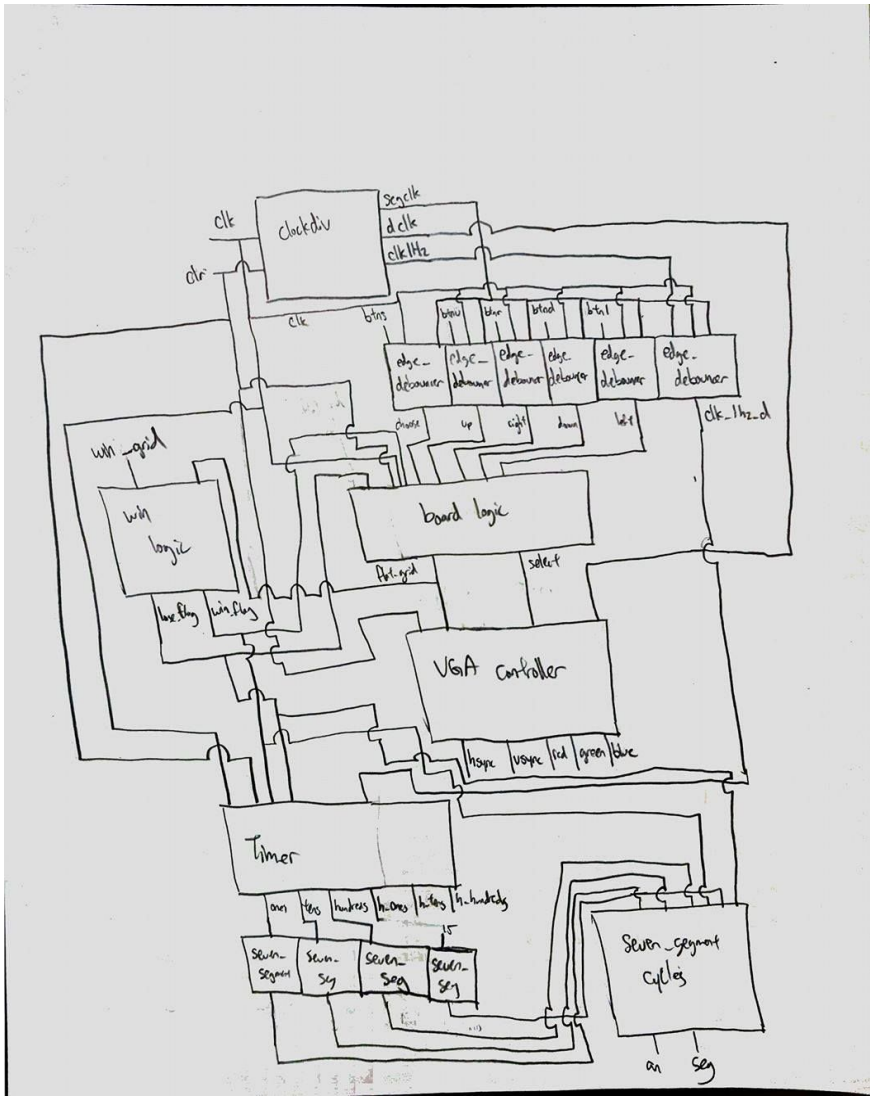
timer

In order to keep track of the time that has passed since the game has started, we use a timer that decrements from 900 seconds. If it ever reaches 0, the player has lost. To implement this, registers are allocated to be initialized to 900 seconds. At every cycle of the 1Hz clock, the time left is decremented by 1 second. If clr is every set to high, the timer is reset to start at 900 seconds and start decrementing again. However, if the player has already won, as indicated by win_flag, the time is left alone allowing the player to see the amount of time they finished with.

seven_segment

This module simply takes in as input a register representing the number we want to output on the seven segment display and uses a case statement to output the corresponding to which cathode segments (CA - CG) should be illuminated.

The following diagram represents the top module, highlighting the relationship between its sub-modules.

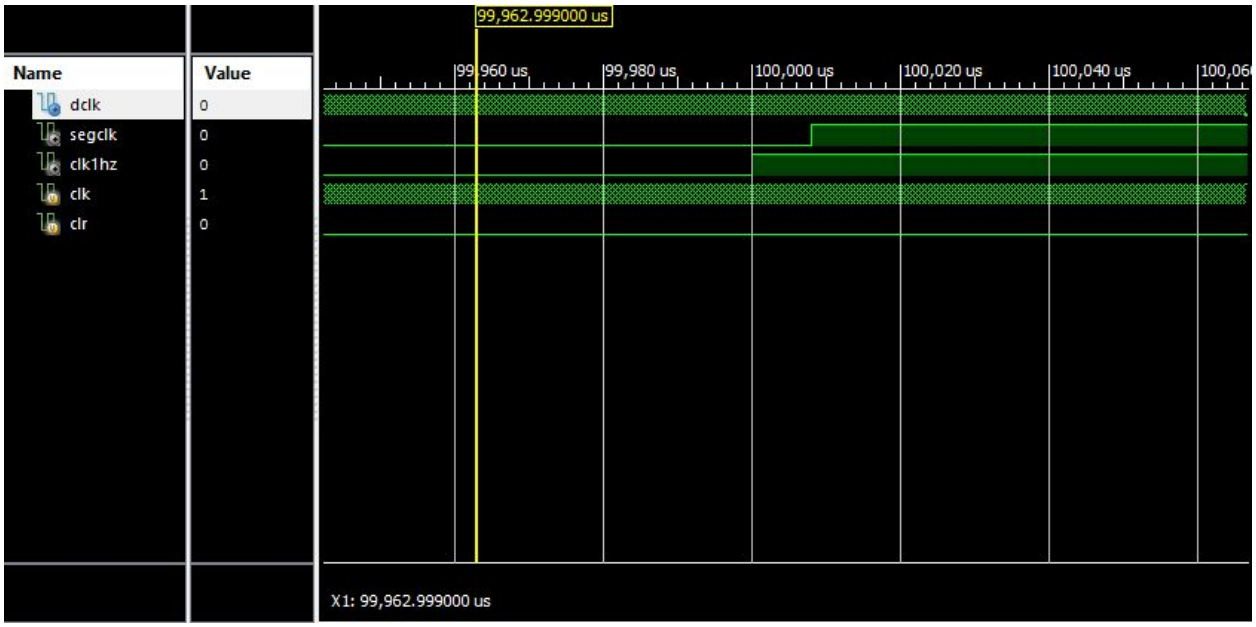


III. Testing and Simulation

Our general procedure for testing was to individually test each module for correctness, and then test the top module to ensure that all the submodules interacted correctly. Again, as in the previous experiment, we ran our clock with a period of 1 ns (so a master clock of 1000 MHz instead of 100Mhz). This was done for the sake of time and convenience; already, it took minutes for the model to simulate even a single second. However, this was not at all a hindrance to the legitimacy of the tests; given that we knew this in advance, we could still test for predicted behavior if the experimental period was 1/10th of what it was ideally supposed to be (e.g. a 1Hz clock having a period of 100ms instead of 1s). We did not test the vga module, as the results from the simulation cannot easily be translated to actual performance on the screen, instead, we relied on visual cues for testing the VGA output.

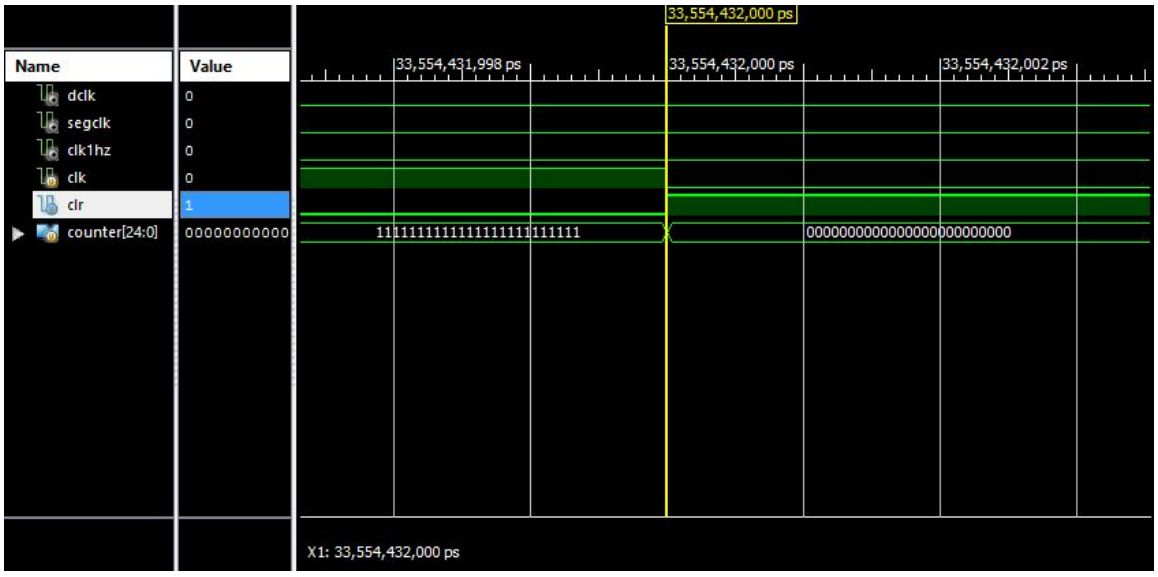
clockdiv

We began by testing the clockdiv module, with the simulation summarized below:



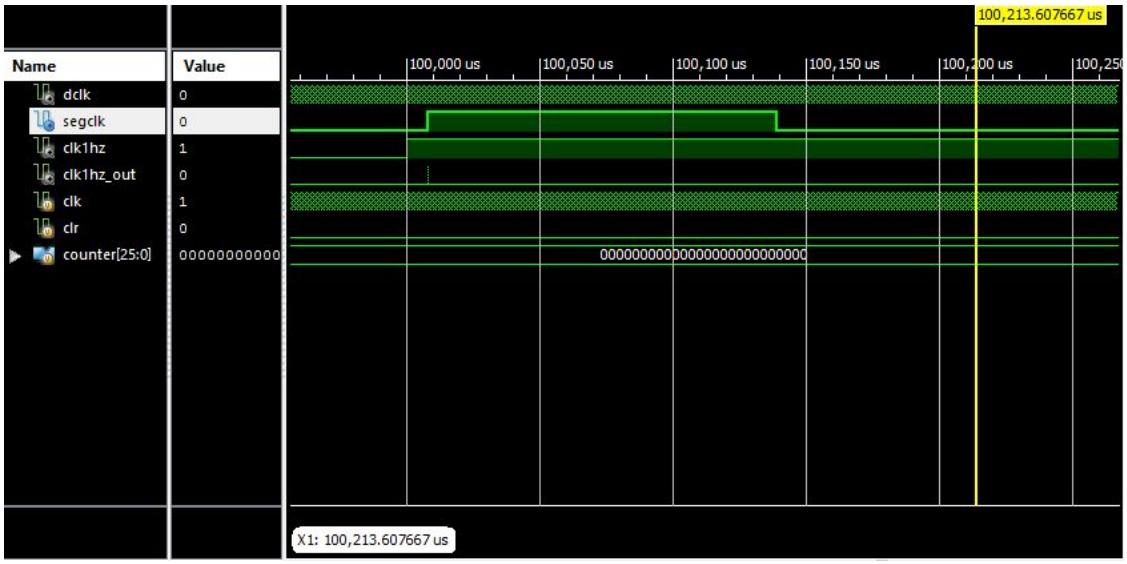
The simulation was zoomed into the 100,000 microseconds (100 ms) mark because this is where we expected our 1 Hz clock to rise. Overall, all the clocks worked as expected. Dclk was supposed to run at at 25 MHz, which we confirm since it has a period of 4 ns (checked by subtracting the times of adjacent rises, for example between 99,967,499 ns and 99,967,495 ns). Segclk had a frequency of 381.47 Hz, so a rise slightly after the 100ms mark makes sense. And, as expected, the 1 Hz clock rose at the 100ms mark.

Next, we wanted to test the clear functionality; this was done using a counter to ensure a delay, after which the clr input was raised (and kept) at high.



The clr was raised around the 33 ms mark, and afterward, all the clocks stayed on low (and never raised), thus ensuring that our clear functionality worked.

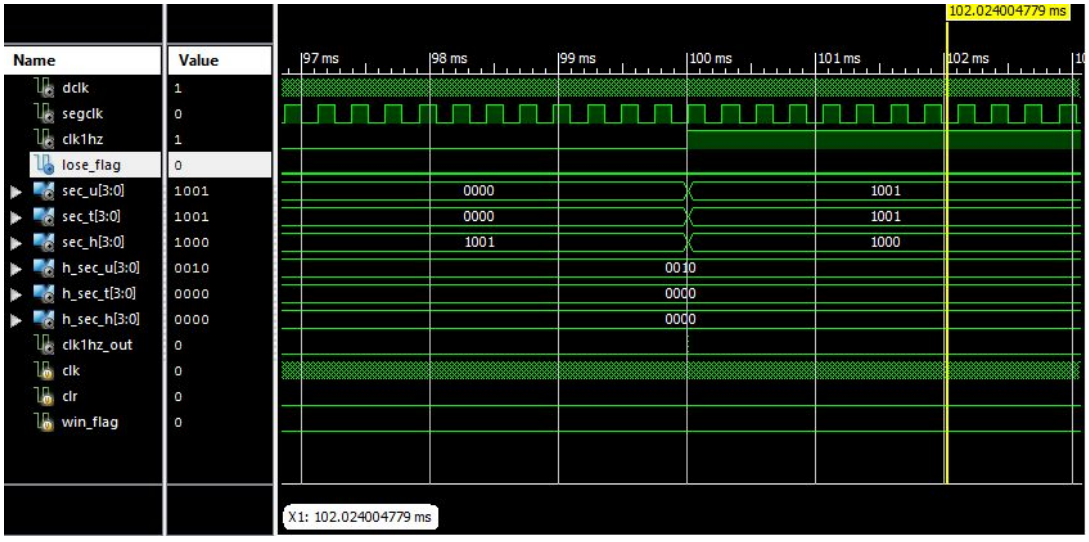
edge_debouncer



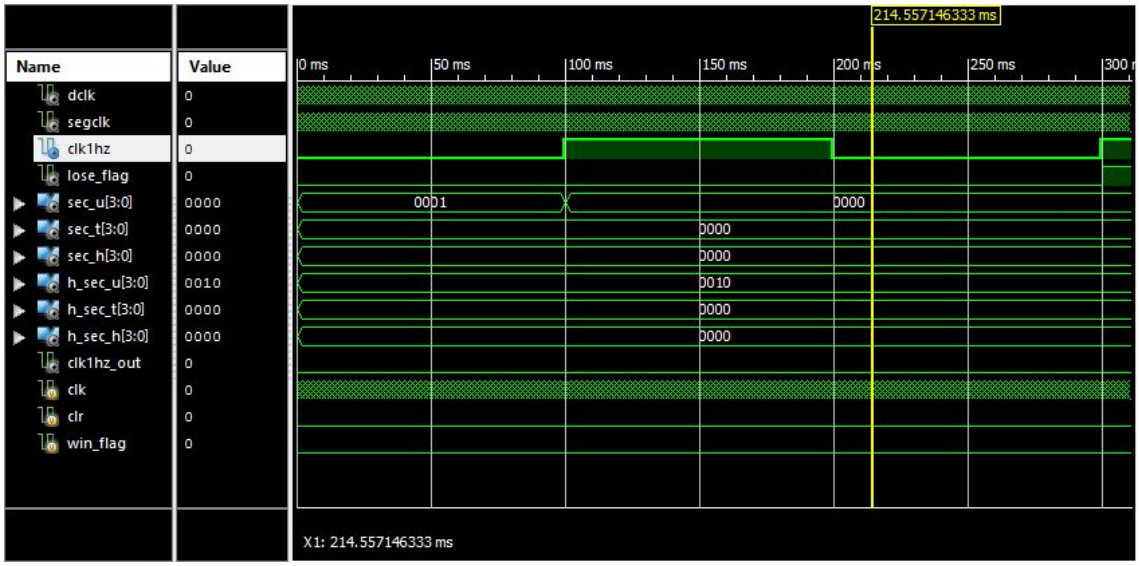
We took the 1 Hz clock as input to be debounced (the reason why is explained in the next section). As indicated by the tiny sliver in clk1hz_out, the debounced clock only rises for a tiny period of time each time the input to the debouncer (clk1hz) rises. This is to ensure that actions that only need to be done once per rising edge actually operate that way.

timer

We tested our timer functionality by calling the properly working clock divider module to generate the clocks, and passing these in as inputs into this module. We also had to call the edge debouncer module, as we wanted our operating to occur only once on the rising edge of the 1 Hz clock (as opposed to every 100 MHz whenever the 1 Hz clock was high).

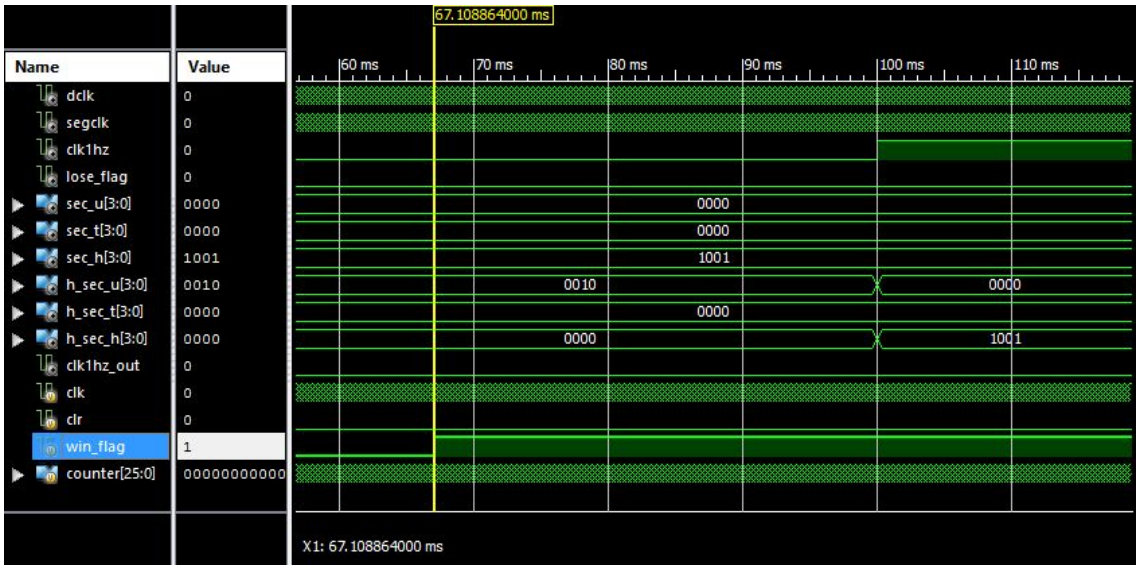


As demonstrated in the simulation, the timer accurately decremented after one second, changing from a value of 900 to 899. This indicates that the hundreds, tens, and units portions to be displayed on the seven segment all function correctly. Next, we wanted to test that the lose_flag updated correctly (when time ran out). However, this would have taken too long for 900 seconds, so we reduced the starting time to 1 second.



One thing that is important to note in the above simulation is that, from the point that the digits hit 000, it takes another rise of the 1 Hz clock for the lose_flag to rise. While this isn't the absolute best solution (it technically gives the player an additional second to win), it is fairly insignificant, and our implementation (where you either lose or decrement) ensures that no undetermined behavior occurs, so this is passable.

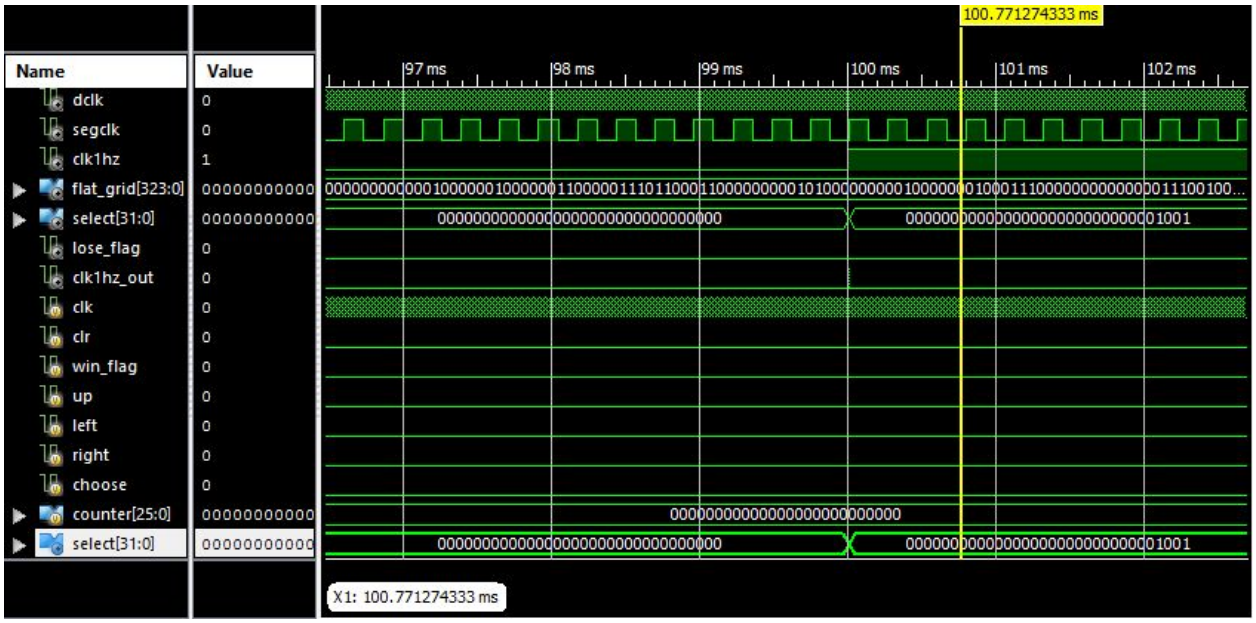
Next, we wanted to check winning and high-score functionality; this was done by incrementing the win flag to one after a delay (through a counter) .



The high score (which was initially set to 2) updates to the value in the unit, tens, and hundreds counters when the win flag is raised, as their values are higher than the values currently held in the high score. Additionally, the counters stop decrementing, but rather stay at the value they were when the flag was raised.

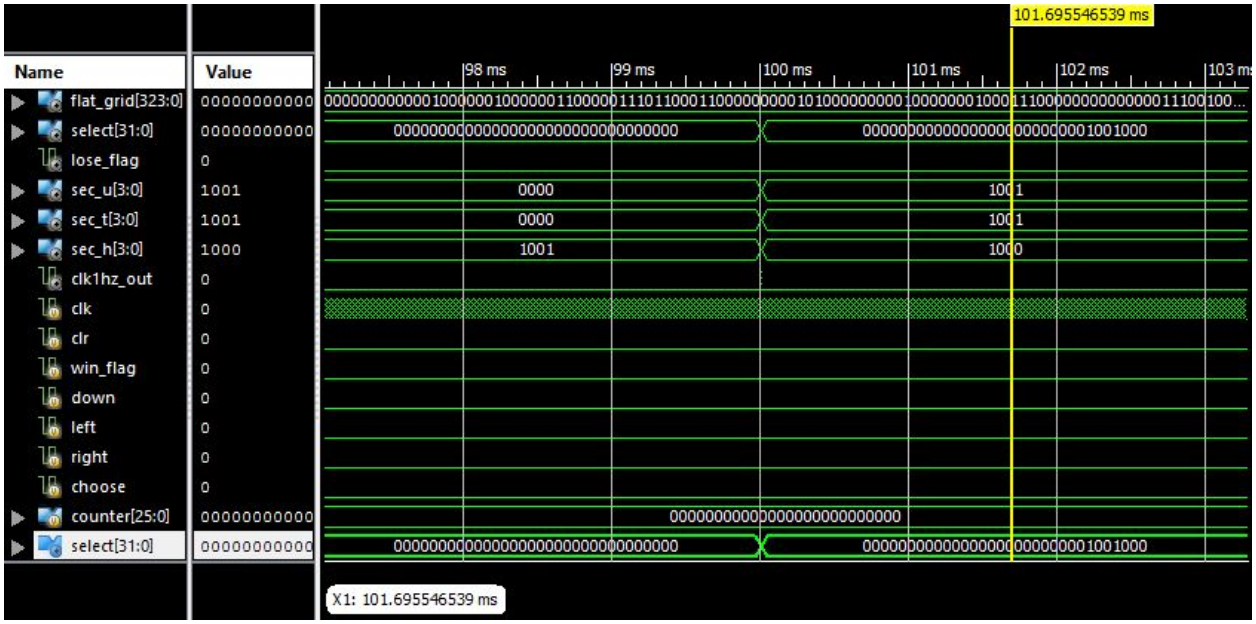
board_logic

Because the board logic deals with actually moving throughout the grid (with the use of the debounced buttons), I had to find a way to simulate that. I decided to do so by using the debounced 1Hz as input to the down button (emulating a move down every second). Select (a 32-bit register that holds a value between 0 and 80, shows the current player position within the grid).



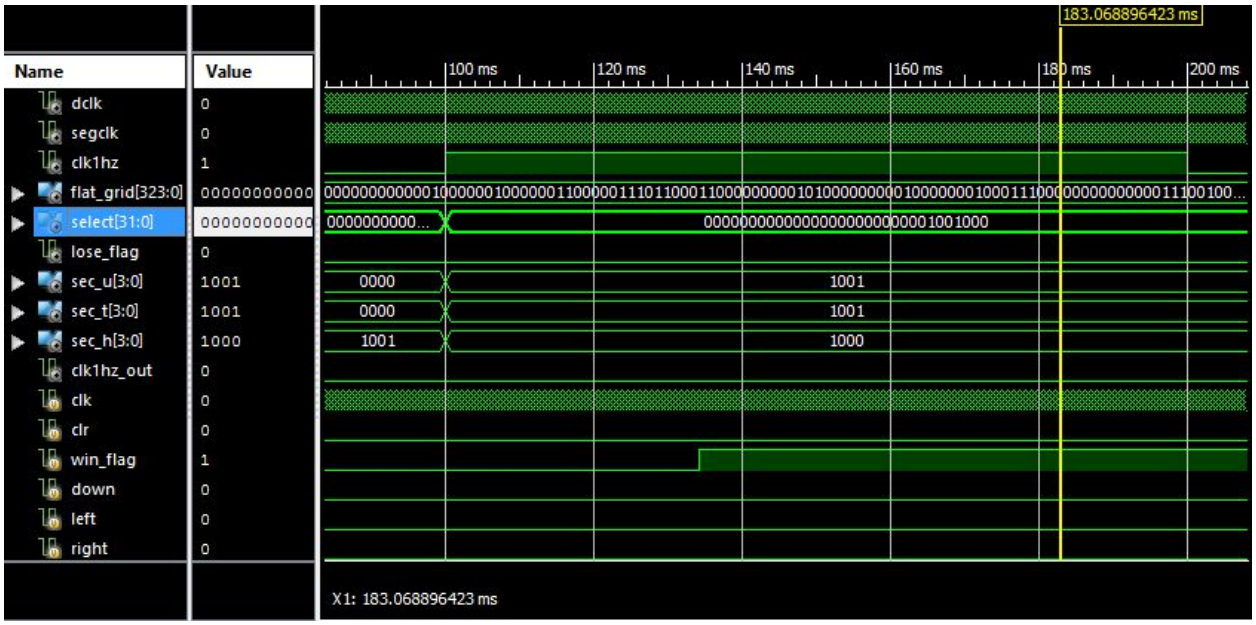
The debounced rise (represented by the small sliver increase in clk1hz_out around the 100ms mark) of the down button triggered a change in the select value to a position one down. It jumped from 0 to 9 because the tiles in the grid are labeled 0-81 starting from the top left, and working left-to-right and then down a row.

Next, we wanted to test boundary detection, we did this by passing the debounced clock to the up button instead; since we start with a select of 0 (indicating the top-leftmost block), a move up should actually loop to the bottom-leftmost tile of the grid.



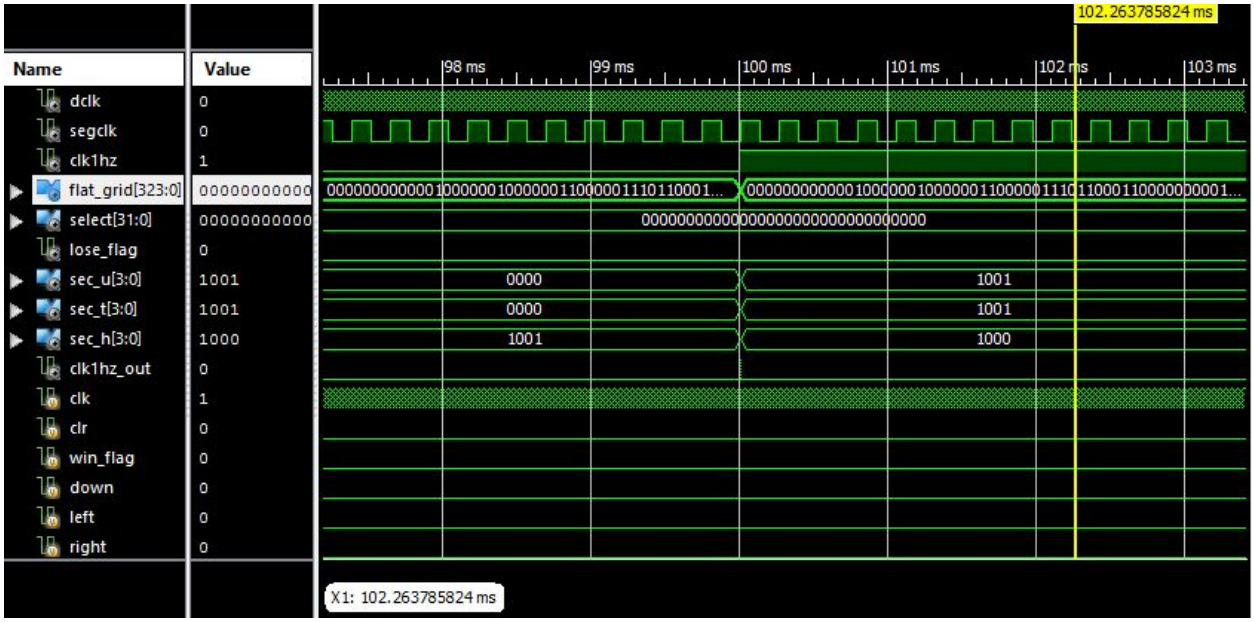
As is evident in the simulation, the select went from 0 to the value 72, indicating that the position accurately looped. This could be tested in all directions, with the same results.

To test the win logic, we implemented the same counter from when we tested the timer that as a result raised the win flag after a delay. The same logic could be used for lose_flag, as they are implemented in the same way; rises in the clk1hz_out will no longer cause any change to select, as the game has ended.



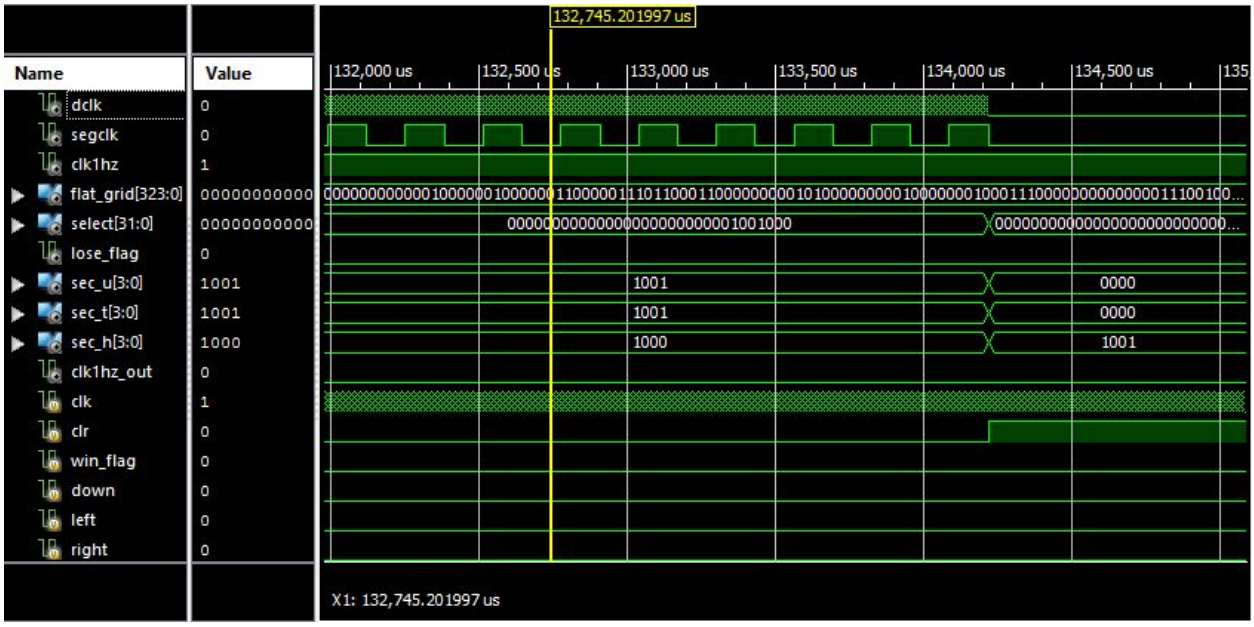
Although select changed prior to the win flag being raised, at the next high cycle of clk1hz_out (indicating a button press) at 200 ms, nothing happened to the select value.

Next, we wanted to test select functionality.



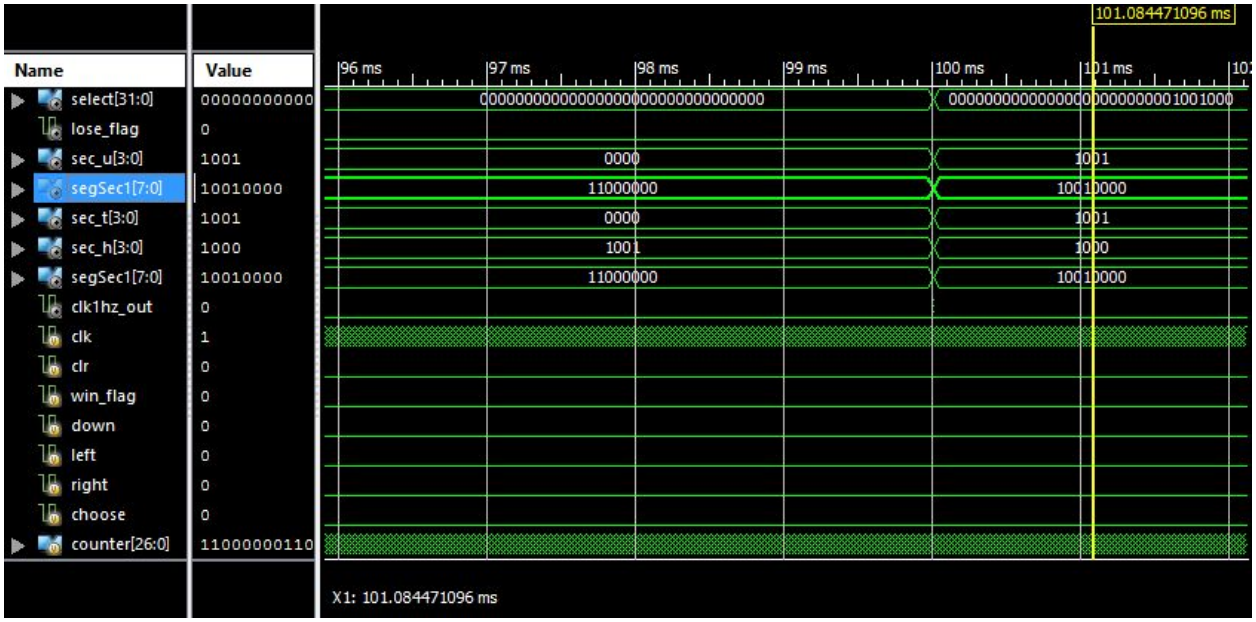
In this simulation, the rise of clk1hz_out resulted not in a change to select, but rather the flat_grid itself. While the results are hard to extrapolate from the image, as flat_grid is an array of 324 bits, the change indicates a modification of the top-leftmost tile in the grid.

Finally, we wanted to test the clear functionality.



When the clear signal was raised to high, the select value, which had been changed, reverted back to its original position of 0 (the top-left tile). If the value of a tile had been changed, that too would have been reset.

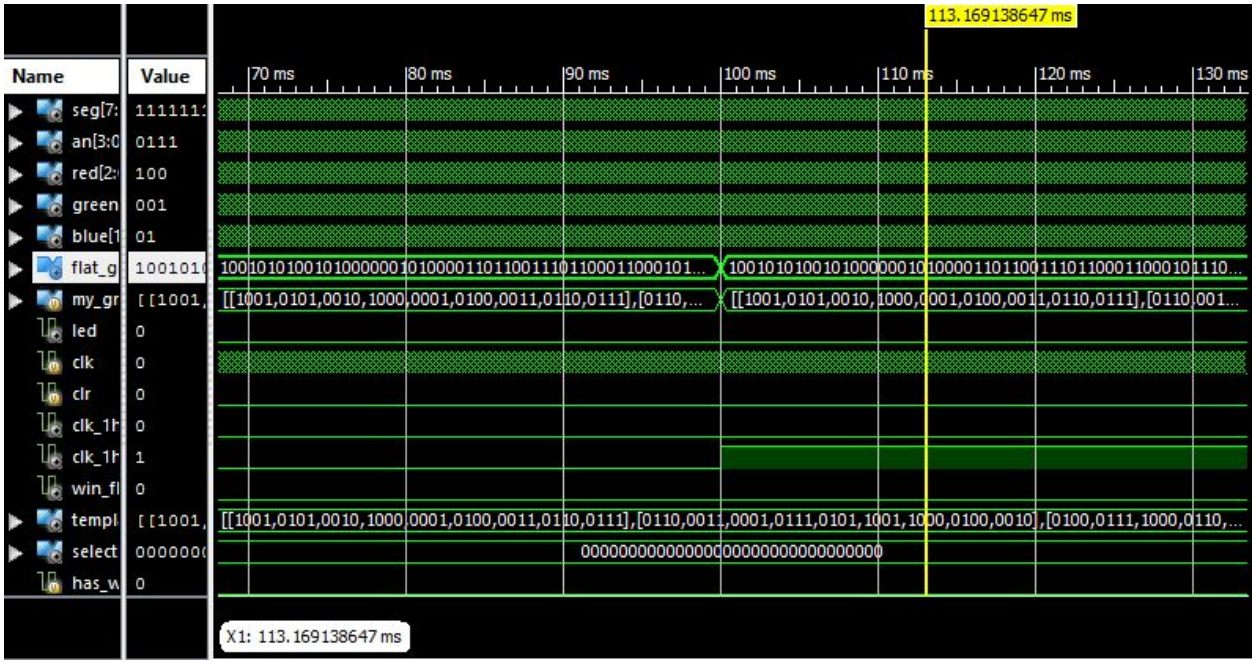
seven_segment



This module is straightforward, simply translating the output of the timer to a 8-bit format representing the cathodes to be lit in the seven-segment. Here, it is highlighted with sec_u and segSec1; when one changes, the other changes as well.

top module

In our top module, we wanted to ensure that all the modules worked together. Previously, when we were testing the win cases in the sub-modules, we were artificially sending a win signal, when actually the win test is implemented in the top module. Thus to test the top module, we set the board up so that every tile was already correctly filled in except for the leftmost tile. Additionally, we had the debounced 1Hz clock as the input to the choose button, so that the value in the leftmost tile would be changed. Therefore, the grid would eventually become solve,thus triggering the win logic and allowing us to confirm that the project as a whole is working.



Because the win is eventually detected, we can confirm that our project is working as expected.

IV. Conclusion

Overall, we were very pleased with the outcome of our project; while there were a few flaws present, overall the game ran very smoothly, with a clean interface through the VGA and seven segment display that as a whole led to an interesting gameplay. Initially, we were going to implement actual numbers to display on the VGA in each box of the Sudoku grid; however, while doing so would still be interesting, we realized from demoing our project that using colors for Sudoku was just as intuitive as numbers. Because the underlying goal was to have unique elements per column, row, and square, it did not seem to matter what these elements were. With that being said, there are a few things we would like to further implement if we had more time. Our puzzle was currently a single preset grid; therefore, when a player reset, he would simply have to play the same puzzle over again. If we had more time, we would have liked to have made a dynamic board; while difficult (the main challenge being to ensure a single solution), the algorithms for doing so have been documented, so our task would largely be to transfer the logic over to Verilog. This would also result in dynamic checks for winning in our code, as opposed to block-per-block confirmation. Another thing we would like to implement if we had more time is error detection. Currently, when a player inputs an incorrect color (one that causes a duplicate value in the same row, column or square), they receive no indication that they made an incorrect move; we would like to implement a detector through a red trim around the incorrect box so that the player immediately understands their mistake; however, this functionality is not vital to the game, and would only be a nice add-on. Finally, unmodifiable blocks (the ones set in the

beginning of the game) are currently indistinguishable from other blocks; if we had more time, we would make some visual change to these blocks to indicate this, as opposed to players attempting to change the block's value with nothing happening. We feel like this project was a good test of everything we learned over the course of the quarter, and we were especially satisfied with learning how to use the VGA, as we can use such knowledge in future project implementations.