

CS M152A: Introductory Digital Design

Laboratory

**Lab #4**

**Tetris**

Name: Kenny Chan, Kenny Luu, Kyle Reidy

UID: 004769092, 104823244, 104839297

Lab Section: 5

Group 8

Due Date: June 10, 2018

**Introduction**

For our final project, our team attempted to recreate a simple version of the classic game of Tetris. Tetris is an electronic game in which falling blocks of various configurations descend from the top of a two-dimensional board. The objective of the game is to accumulate points by filling every space within a row or multiple rows. The game is lost when the blocks fill to the top of the board. In our simplified implementation, we chose not to include features found in many versions of Tetris, such as stages, the number of lines cleared, or showing the next falling block, which are displayed in figure 1.

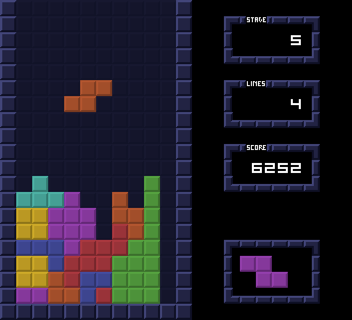


Figure 1: Example board of a game of Tetris

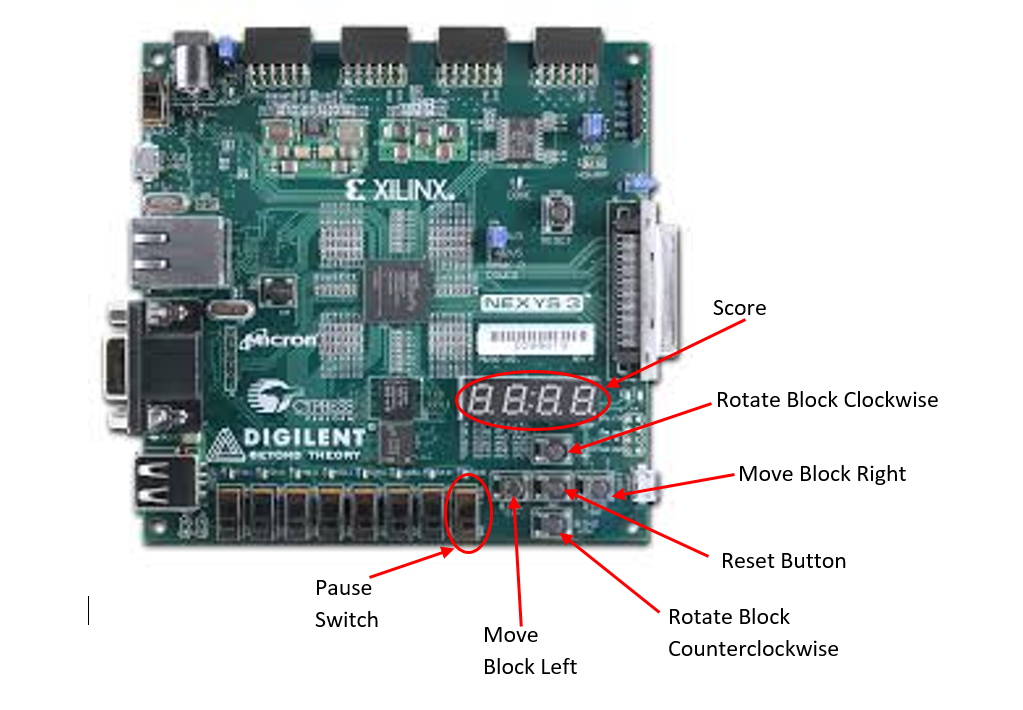


Figure 2: Functionality of buttons for our implementation of Tetris on an FPGA board

We used VGA to display a 10x20 board on a monitor, and the FPGA board to display the score and control the game. In particular, we used the left and right buttons on the FPGA board to move the falling block accordingly. The up and down button were used to rotate the piece clockwise or counterclockwise. To reset the game, we used the center button, and to pause, we implemented a switch.

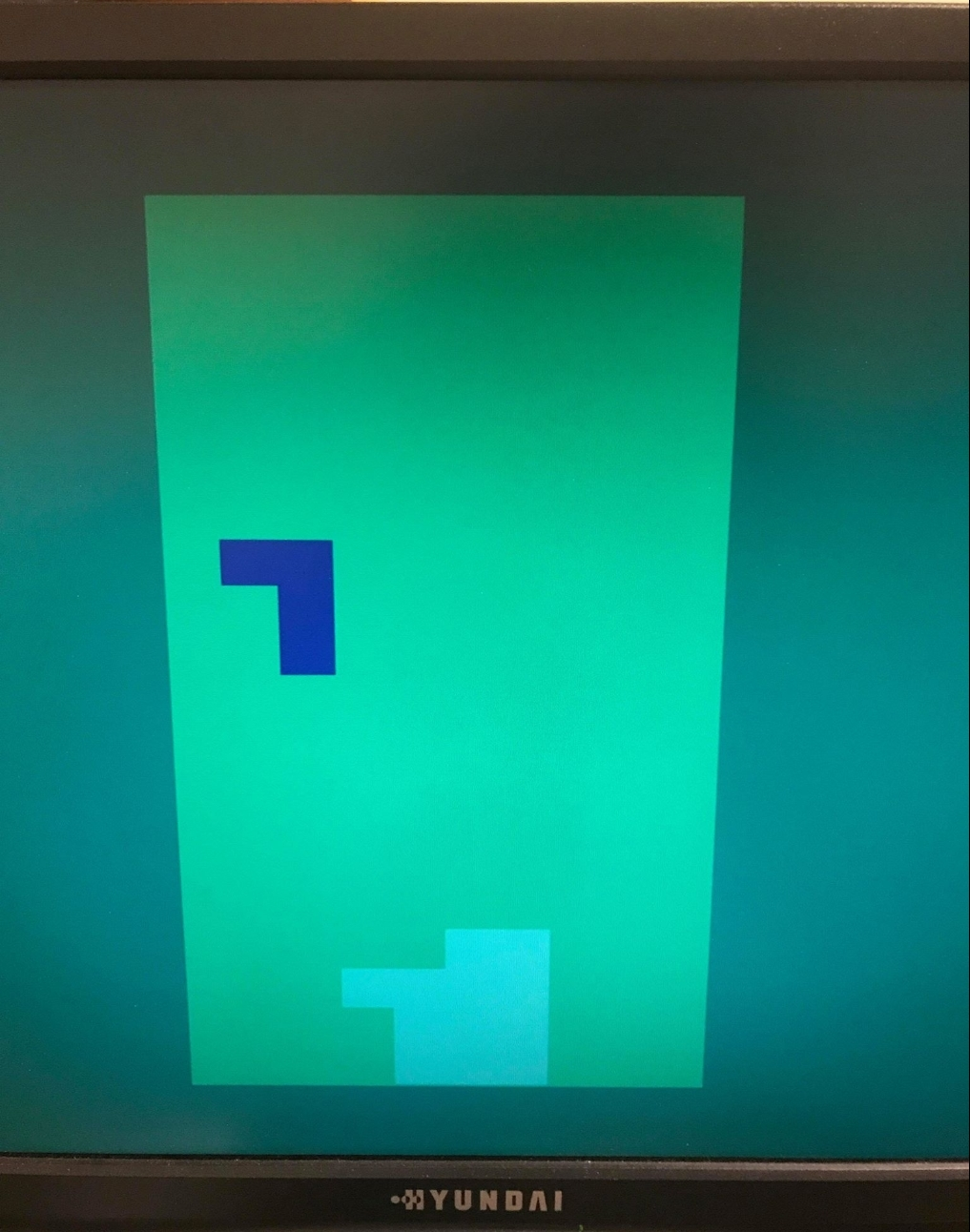


Figure 3: Photo of our 20x10 Tetris board, with a falling block (blue) and multiple static blocks (teal)

**Design Description**

For our design, we used one main top module that utilizes six submodules: a clock divider, game controller, board, vga controller, score displayer, and button debouncer.

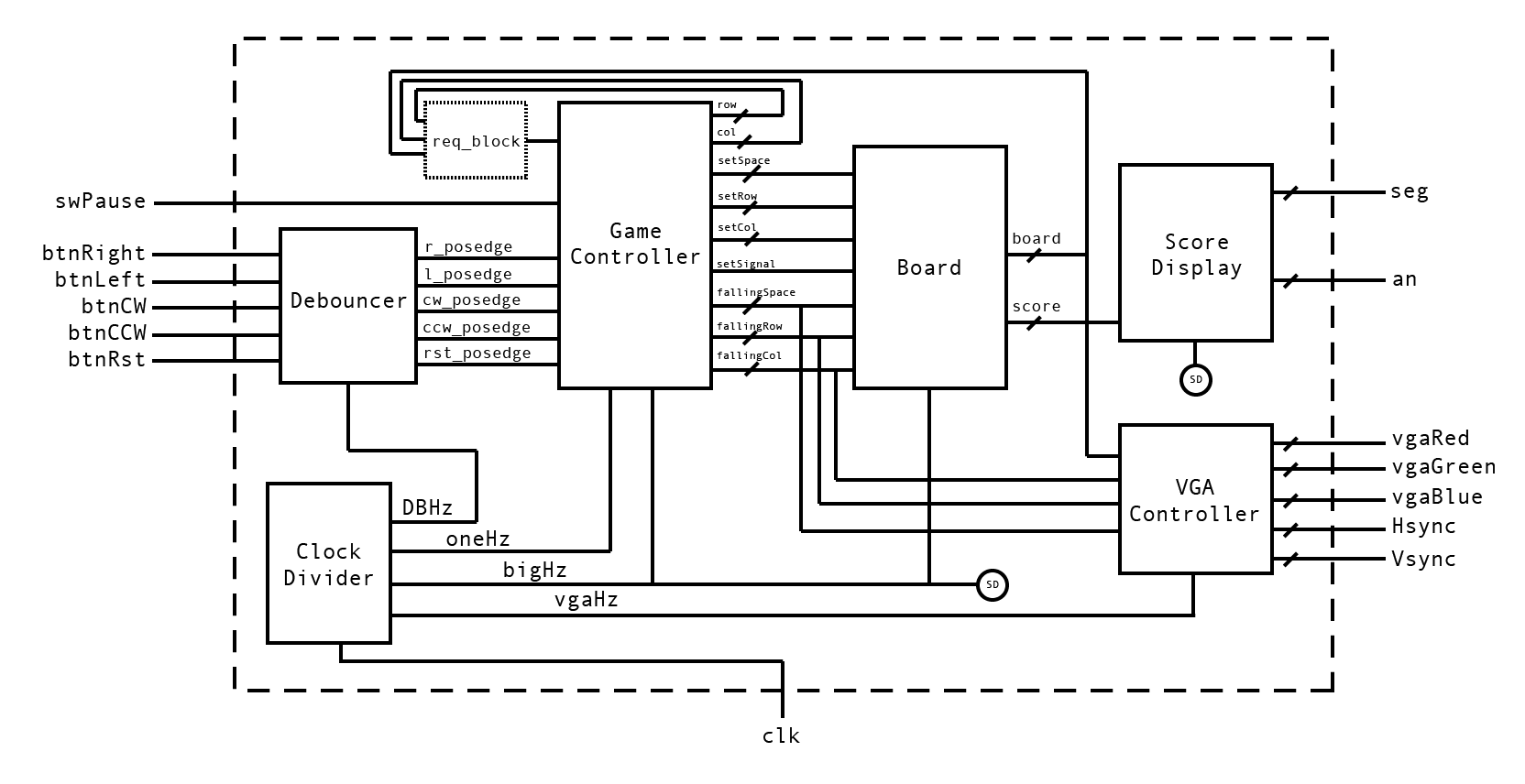


Figure 4: Overall design with its six submodules.

Note: req\_block is not a real module but rather a signal generated by the top module through combinational logic.

**Clock Divider**

Like our previous labs, the purpose of the clock module is to create clocks of various frequencies derived from the FPGA master clock. To create such clock, we added a unique counter for each clock at the posedge of the master clock. Once the counter reaches a count corresponding to the number of 100 MHz pos edges within that frequency, we invert the clock and reset the unique counter. The value can calculated using the formula:

, X = desired clock rate, V = value at which to reset the counter

For example, we want to invert the one Hz clock for the rate at which Tetris blocks fall. The counter must be inverted twice every second to represent a full cycle of a one Hz clock. There are 100,000,000 Hz in 100Mhz and we want to invert the one Hz clock so we will reset the unique counter at 50,000,000, and invert the one Hz clock. This process is repeated for each of the other clocks.

**Game Controller**

The idea behind our design was to have the game controller module take in the user inputs, such as moving left or right, and control the dynamic falling blocks. The board module would detect blocks that have reached the bottom-most row and save them onto a static block board. The feature to set dynamic blocks into static blocks if they have fallen on top of other static set blocks was not implemented in time. By separating the game components into falling and static blocks, it was easier to manage the game’s logic and the VGA display.

The Game Controller module manages the random generation of dynamic blocks and their movements based on user inputs, along with determining whether they have reached the bottom-most row. It does this by taking in a 500Hz and 1Hz refresh signal from the Clock Divider along with the buttons and switches from the Button Debouncing module. The 500Hz is used as the refresh signal for the module and the 1Hz signal is used to lower the dynamic blocks down by one row. If the move left or right button signals are detected, then it will move the blocks left or right accordingly based on the 500Hz signal. The feature to rotate the blocks was not implemented in time.

To determine whether dynamic blocks cannot be moved left or right (if they are along the edge of the board) or if they have reached the bottom of the board, the Game Controller module outputs a row\_out and col\_out signal to check whether the movement is valid. The Tetris top module checks the row and col with the static board outputted by the Board module to determine whether it is valid. Based on the result, the req\_block input is set and the Game Controller will either move the dynamic blocks, not move them, or determine that they meet the requirements to be set as static blocks.

The Game Controller has many other outputs, such as setSignal, falling\_space, falling\_row, falling\_col, set\_space, setRow, and setCol. All of these outputs are used by the board module and VGA module to determine the shape and location of the dynamic falling blocks. If the dynamic blocks that are currently falling do not meet the requirements to become static blocks, then the VGA module will use these outputs to display them in the appropriate location. If they do meet the requirements to become static blocks, they the board module will use these outputs to append them onto the main tetris board. The Game Control will then randomly generate a new dynamic falling block and the process will repeat.

The random generation is done through multiplication of the block’s characteristics, which “simulates” randomness but is not truly random. The possible blocks that can be generated are shown below.

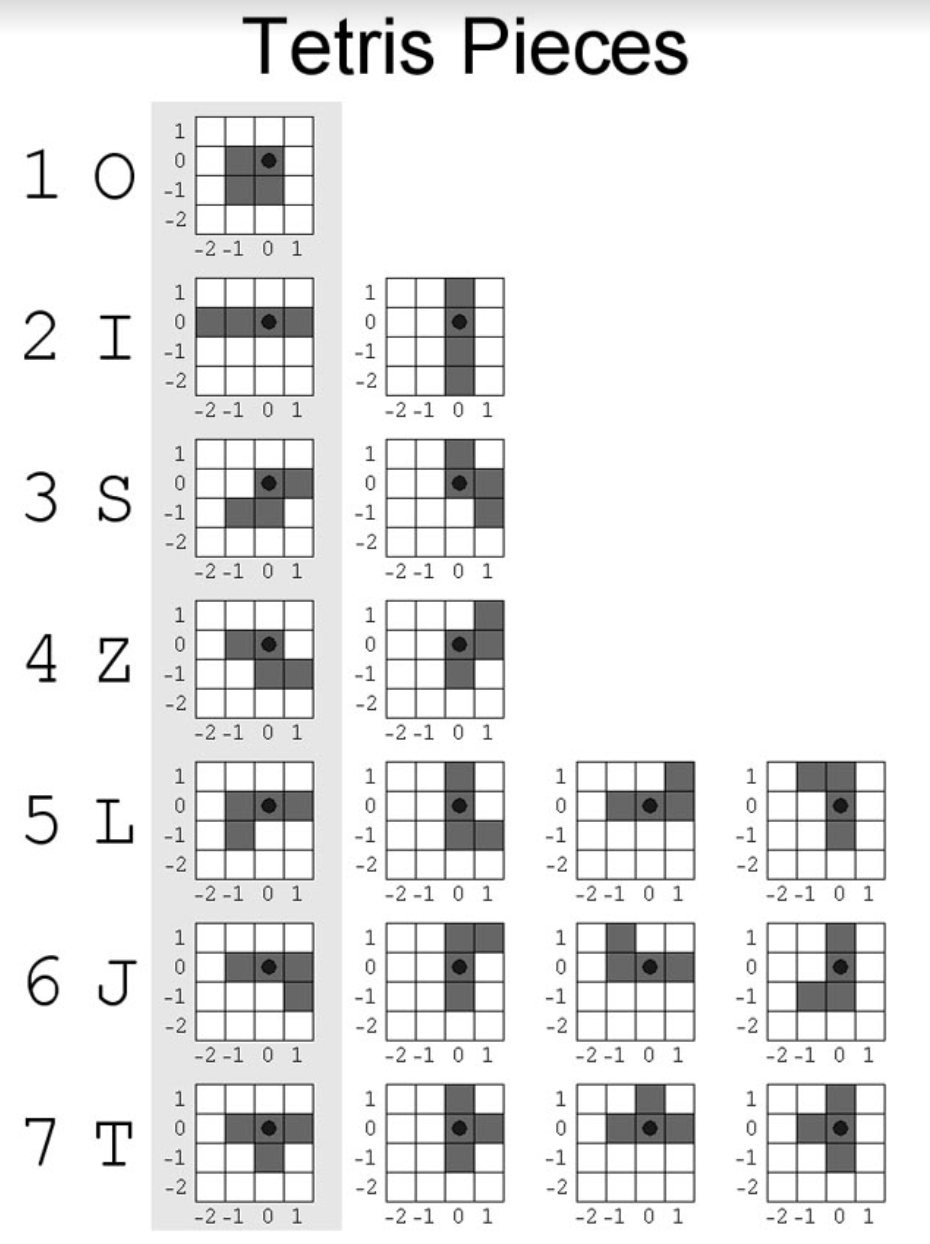


Figure 5: List of possible tetris block orientations that can be generated

If the reset signal is detected, it resets all the states back to 0 and generates a new random block. If the pause signal is set high, it locks the dynamic falling block in place.

**Board**

This module managed the static set blocks (the actual 20x10 tetris board) and the player’s score.

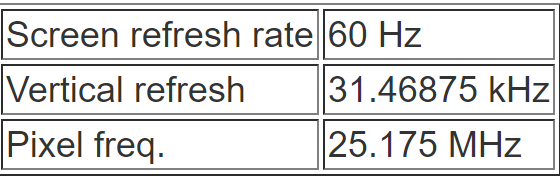
It took in a refresh clock rate set to 500Hz by the Clock Divider module that it would use to refresh its data. It also has a setSignal input, which would be set high by the Game Controller module if the dynamic blocks reached the bottom row (the feature to drive setSignal high when dynamic blocks fell on top of static blocks was not implemented in time). It also has inputs for row, col, setSpace, setRow, and setCol. These signals are generated by the Game Controller module whenever setSignal is high. If the dynamic blocks in the Game Controller meet the requirements to become static blocks in the Board module, setSignal is driven high and the rest of the inputs tell the Board module where to add the new static blocks to the main tetris board. The Game Controller module can then move onto the next set of dynamic falling blocks and the process repeats.

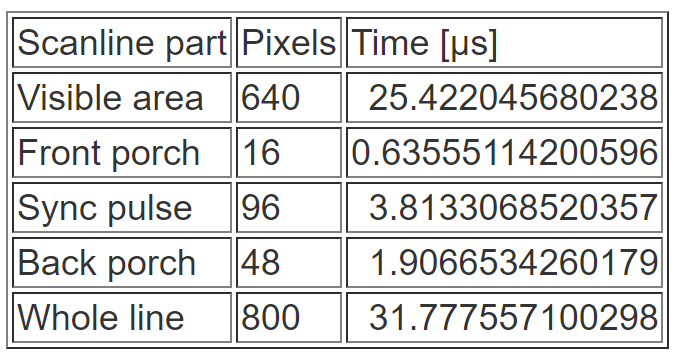
Since the Board module is incharge of the full tetris board, it checks each row of the board on every refresh clock cycle to see if it is full. If a row is full, it increments the player’s score, shifts all the blocks above that row down by one, and clears the topmost row. Doing so would have the same effect as deleting the full row and shifting everything above it down by one.

If the reset signal is set, all the states including the main board are set back to 0.

**VGA Controller**

The VGA Controller module handles all the timing and signals needed to display the tetris board onto a 640 x 480 VGA monitor. It does this by taking in a 25Mhz signal from the Clock Divider module, processing all the time requirements, and displaying the appropriate colors to the monitor at the appropriate times. The VGA Controller module is broken down into 3 submodules that handle the vsync timing, hsync timing, and color selection. The color selection is based on the inputs it receives from the Board module and Game Controller Module, along with the timing to determine whether the current pixel is within or outside the tetris board’s bounds.

****

****

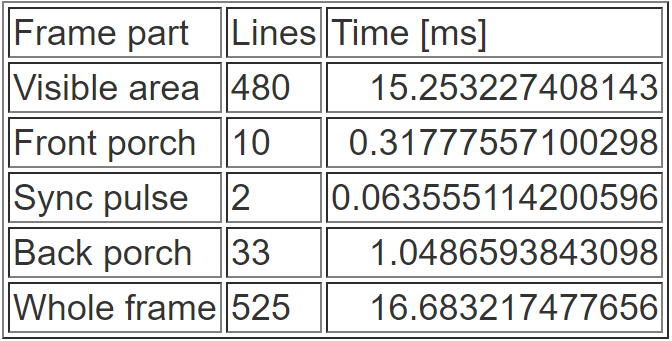
****

Figure 6: Timing specifications to generate 640x480 images onto VGA displays

**Score Display**

To keep display the score, a four digit decimal number, represented 15 bits, is passed into the score displaying module. This module’s responsibility is to convert the bits into four digit to be presented on the FPGA’s seven segment display. To compute the least significant ones’ place digit, we took the 15 bits and performed modulo operation by 10. For the other digits, we manipulated the score using similar modulo operations, which can be seen in the table.

|  |  |
| --- | --- |
| Place Value of 15 Bit Score | Digit Corresponding to Place Value |
| Ones’ Place | score%10 |
| Tens’ Place | (score%100 - score%10)/10 |
| Hundreds’ Place | (score%1000 - score%100 - score%10)/100 |
| Thousands’ Place | (score - score%1000 -score%100 -score%10)/1000 |

We then converted each of these digit to the corresponding 8 bit values that represent which segments of the display to light up. Because only a single digit of the seven segment display can be displayed at a time, we used a high frequency to iterate through each of the digits rapidly. The seg[7:0] from the UCF file correspond to the 8 bits, and the selector bits, an[3:0], correspond to which digit to display.

**Button Debouncer**

The debouncing module takes in a declock signal from the Clock Divider module, along with the state of the reset, left move, right move, counterclockwise rotate, and clockwise rotate buttons. The debouncer works using the same method as lab 3’s debouncer. At every rising edge of the 10Hz signal, it stores the state of the buttons in their own 3 bit register concatenated with the state of the previous 2 cycles. It determines whether a rising edge has occurred by checking for a transition from 0 to 1. The module then outputs a signal that represents the rising edge of the buttons, it is driven to 1 when it sees a rising edge and is pulled down to 0 otherwise. For the reset signal, the output is passed directly to the Counter module.

**Simulation**

**Clock Divider**

To test the Clock Divider module, we used the same test as Lab 3’s clock divider. We set the input to alternate between 1 and 0 very rapidly, which simulates a clock signal. After that, we looked at the waveforms to check whether our clock signals were alternating appropriately. For example our bigHz (500Hz) output would alternate once every couple thousand of normal clock signal rising edges. Our 10Hz output should alternate once every couple hundred of bigHz rising edges, our 1Hz output should alternate once every two rising clock edges of our 2Hz output, etc.

|  |  |  |
| --- | --- | --- |
| **Test Case** | **Result** | **Pass/No Pass** |
| while (1) begin  clk = ~clk;  #1;  end | Output clocks were output at the right frequency based on simulation.  Successful demo further demonstrates it’s accuracy. | Pass |

**Game Controller**

We first tested whether the dynamic blocks would drop down one row at a rate of 1Hz. To do this, we alternated our refreshClock and oneHz inputs and checked to see if col\_out decremented by one each time oneHz was a posedge.

We then tested whether the dynamic blocks could move from left to right appropriately. We alternated the refreshClock input signal and set our move\_Left or move\_right input to high. We checked whether the block moved correctly by checking the row\_out and col\_out output signals.

We used the same approach to test the rest of the behavior.

After that, we checked whether invalid movements would be blocked.

Then we tested if the blocks would be set if they reached the bottom.

Then we tested the rotations of the blocks.

Then we tested if the reset and paused worked.

All tests were done with a board set to 200’b0 and the req\_block signal set to the following-

assign block = board[row\*10+col];

|  |  |  |
| --- | --- | --- |
| **Test Case** | **Result** | **Pass/No Pass** |
| refreshClock= ~refreshClock;  #Alternate Rapidly  oneHz = ~oneHz;  #Less Rapidly  isPaused = 0;  reset = 0; | This tested whether blocks would drop by one row each second.  The col\_out signal decremented appropriately, no other signals changed, which is what is expected. | Pass |
| refreshClock= ~refreshClock;  #Alternate Rapidly  oneHz = 0;  isPaused = 0;  reset = 0;  move\_L = 1;  #1;  move\_L = 0;  move\_R = 1; | This tested whether blocks could move left and right.  The row\_out decreased by one at first (indicating a left movement) and then incremented by one (indicating a right movement) | Pass |
| refreshClock= ~refreshClock;  #Alternate Rapidly  oneHz = 0;  isPaused = 0;  reset = 0;  move\_L = ~move\_L #20 times | This tested whether blocks would not move if the move is invalid.  The row\_out decreased until it dropped down to 0. After that, row\_out remained at 0 and did not change. | Pass |
| refreshClock= ~refreshClock;  #Alternate Rapidly  oneHz = ~oneHz;  #Less Rapidly  isPaused = 0;  reset = 0; | This tested whether blocks that have reached the bottom of the board would become static blocks.  The setSignal output was change from 0 to 1 when col\_out was 3 (indicating that it has reached the bottom row).  The outputs falling\_t, falling\_space, falling\_row, falling\_col, set\_space, setRow, and setCol we all changed to their appropriate values.  The outputs row\_out and col\_out were reset, indicating a new block is falling. | Pass |
| refreshClock= ~refreshClock;  #Alternate Rapidly  oneHz = ~0;  rot\_L = 1;  #1;  rot\_L = 0;  rot\_R = 1;  isPaused = 0;  reset = 0; | This tested the rotation behavior.  No changes, rotation does not work. | Not Pass |
| refreshClock= ~refreshClock;  #Alternate Rapidly  oneHz = ~0;  isPaused = 1;  #4;  isPaused = 0;  #1;  reset = 1; | This tested the reset and pause features.  When isPause was set to 1, the row\_out and col\_out did not change, indicating the block stopped.  When isPaused was set back to 0, the continued changing appropriately.  When reset was set to 1, row\_out and col\_out was reset back to original. | Pass |

**Board**

Using the same methodology as the Game Controller module, we alternated the clock inputs and checked the outputs.

The Board module was easy to test, since it is only changed whenever setSignal is driven high to indicate that dynamic blocks are becoming static and need to be added to the main board.

We also tested the reset behavior.

|  |  |  |
| --- | --- | --- |
| **Test Case** | **Result** | **Pass/No Pass** |
| refreshClock = ~refreshClock;  #Alternated Rapidly  setSignal = 1;  setRow = 2;  setCol = 2;  setSpace = 16’b0000000011110000  #Characteristics of a horizontal L block | The output board was set appropriately, the horizontal L block was added in the correct position and orientation. | Pass |
| refreshClock = ~refreshClock;  #Alternated Rapidly  setSignal = 1;  setRow = 2;  setCol = 2;  setSpace = 16’b0000000011110000  #1;  reset = 1; | The output board was set appropriately, then reset back to 200’b0 when reset was driven high. | Pass |

**VGA**

Testing the VGA would be too difficult to do with the simulation, since we would have to wait through a couple thousand clock cycle alterations before the first pixel of the tetris board would be displayed.

Instead, we tested this by uploading onto the FPGA board and using the lab’s vga monitors to check the output. Since our VGA module was prewritten by one of our group members and ported over to this project, we had no difficulties getting it to work the first time.

**Seven Segment Display**

To test the seven segment display, we used the same test bench from Lab 3, without the adj feature. We used the same set up by alternating the bigHz signal. We also set the counter inputs (which would represent our score) to a static value to see whether the seg[7:0] output matches these static values appropriately.

|  |  |  |
| --- | --- | --- |
| **Test Case** | **Result** | **Pass/No Pass** |
| Alternate only bigHz;  counter0 = 4;  counter1 = 3;  counter2 = 2;  counter3 = 1; | The an[3:0] output successfully alternated sequentially from 1110, 1101, 1011, and 0111;  The seg[7:0] output successfully matched the counter value for the appropriate seven segment display;  (Figure 6) | Pass |

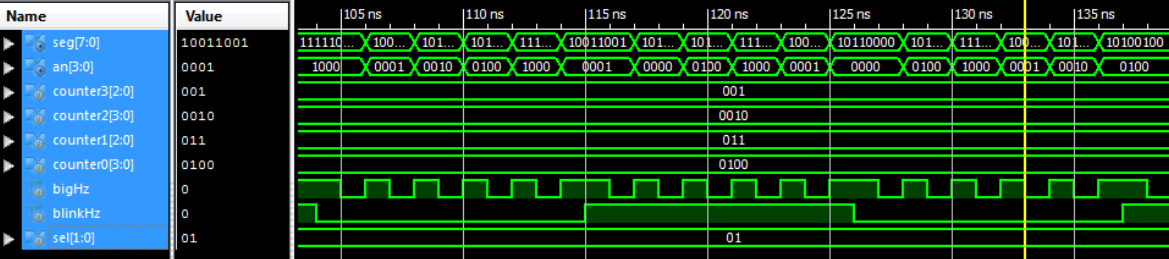


Figure 6: Testing the seven segment display signals

**Debouncer**

Debouncer was tested by uploading to the board and testing button response.

**Conclusion**

In the end, most of our submodules were successful, but a few of them were not. The clock module was able to successfully produce clocks of similar frequencies which was apparent in the simulation. Most of the functionalities of the game controller, such as generating random blocks that fall, left and right movement, pausing, and setting the blocks at the bottom worked. However, we were not able to implement block rotations and we are unsure why blocks would not stack on top of each other. The board module successfully cleared lines and reset the board state, but may also be responsible for blocks falling through each other. The VGA was successful in displaying the board. The score display recorded the score, but began the score count at 7000. Lastly, the debounce module was successfully, and buttons worked correctly.

Overall, we underestimated the difficulty of this project. While we did achieve success with the VGA display, we were unable to stack blocks on top of each other; thus, the game could not be lost. If attempting this project again, we would spend more time experimenting with different ways of representing the board and falling blocks and with various methods of communicating the status of the board between modules. Nevertheless, we gave our best effort and learned a lot about VGA displays as well as writing code for and the intricacies of FPGAs.