Lab 4: Tetris

CS M152A

Group: Jeannie, Cell, Dilshan

Date: 12/12/2018

TA: GU, HONGXIANG

1. Introduction

In this project, a simplified game of tetris was simulated on the Nexys3 FPGA using a VGA port as the display and the buttons and switches on the FPGA itself as the controls. The game consists of a 10x20 block board space in which a singular tetromino falls until it first collides with a surface under it which makes it freeze into place. Each tetromino piece consists of 4 blocks put together into various shapes. This project, however, only makes use of only 3 possible tetromino shapes instead of the usual 7, thus it is a simplified version of the original game. The tetrominos used in this implementation are the line, the box, and the t-shaped pieces designated as the I, O, and T shapes respectively. The player is able to move the tetromino left or right, rotate it 90°, and even make it fall faster as it is falling, but they cannot make it go back up. After a tetromino falls to a surface, a random new piece appears at the top of the board that the player must control again in the same fashion. In order to score points in this game, the player must strategically place the tetrominos in a way that would fill whole rows of the board space with tetromino block pieces. Once a whole row is filled, the row is deleted and a point is assigned to the player for each row that was cleared. Blocks that were placed above the cleared rows are lowered into the first uncleared row under it or, if there are none left, the bottom of the board. The game continues in this fashion indefinitely until the player loses the game, in which the game will freeze until it is reset. The player loses the game when a tetris piece settles on a surface while it is still touching the very top of the board. Other functionalities include the ability to pause the game at any time and the ability to make a tetromino piece fall instantly to the first surface under it. The player controls and reset button are all mapped to the five buttons on the FPGA board while the pause and instantly fall functions are mapped to two of its switches. Points are displayed on the FPGA's seven segment display located directly on the board.

2. Design

The overall design consists of a top level module called "simple_tetris.v" that maintains the registers that will be used to output to the screen. In the following section we will explain how each module behaves.

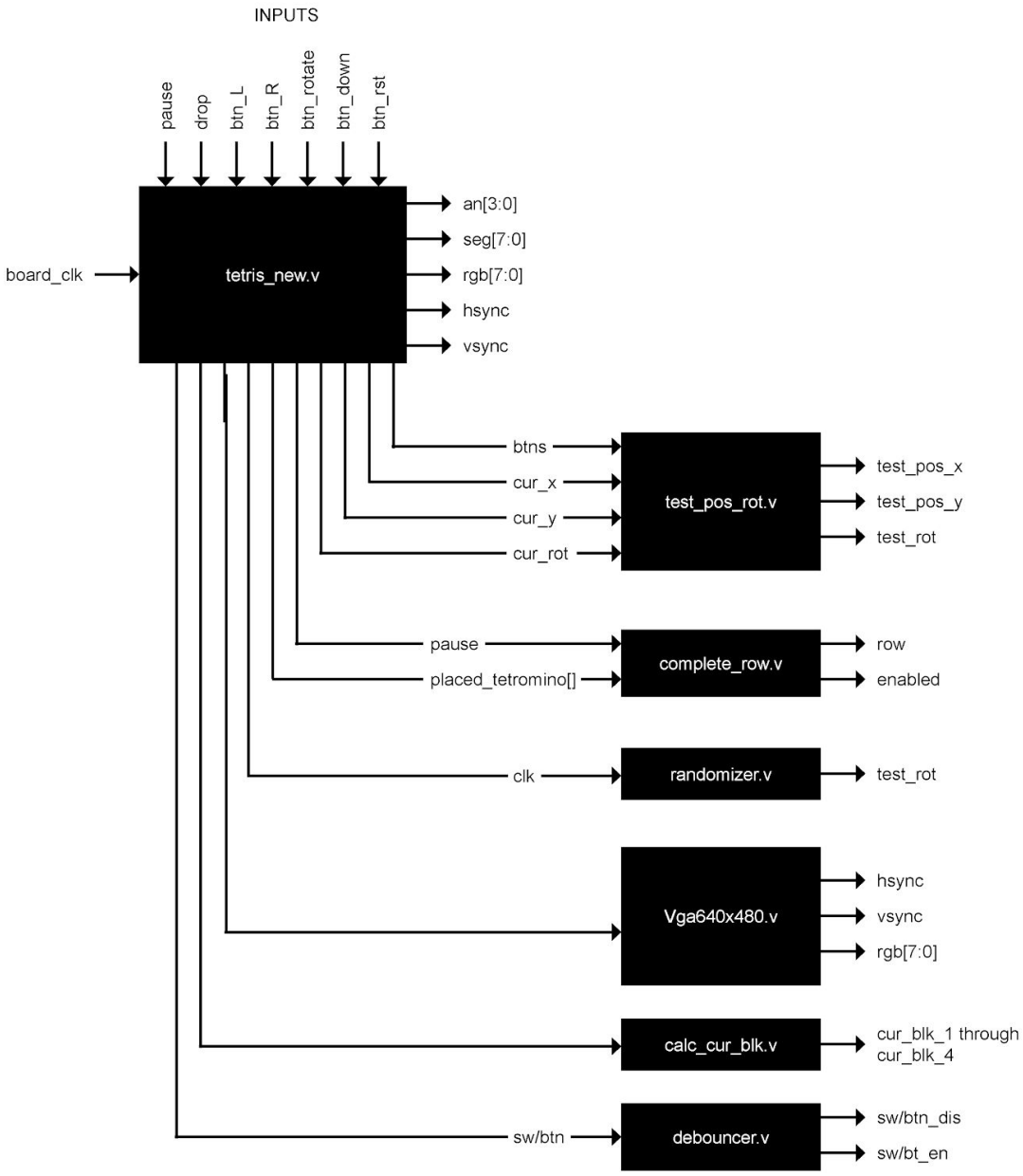

Figure 1. Overview of our 3 Piece Tetris Implementation

2.1 Simple_tetris.v

This is the highest level module, taking five buttons and two switches as input, and outputs to the VGA pins and seven segment display. Specifically there are two switch inputs, *drop* and *pause*, that enact exactly what they're name implies. In the case of drop, the piece will immediately drop to the bottom of the board. There are 5 modes the finite state machine loops through:

Mode	Purpose
MODE_PLAY	Listen to user input
MODE_DROP	Move the piece down until it hits the bottom
MODE_PAUSE	State to identify when user has paused game
MODE_IDLE	Initial state so game doesn't start right away
MODE_CLEAR	State to clear filled rows

Table 1. The Five FSM States and their Functions

The main issue this module tackles is how to represent the 10x22 board in a way that could be used an input to other modules such as **vga640x480.v** in order to represent placed pieces. We ended up using a flattened array to represent the placed pieces, *placed_tetrominos[]*, with a length of BLOCK_WIDTH x BLOCK_HEIGHT, to represent every block on the board. Consider that we have the x and y block coordinates of a piece, then the place in the array for the first piece's block would be in index

$$\text{Index} = (y \times \texttt{BLOCK\_WIDTH} + x)$$

Eq. (1)

In this way, if BLOCK_WIDTH = 10, the first row of the board is stored in elements [9:0] and the nth row would be stored in elements [(10n-1):(10*(n-1)] for 1 <= n <= BLOCK_HEIGHT. On top of the coordinates, this module also keeps track of the 4 possible types of pieces and its rotation. The 4 types of pieces include the I, O, T, and an "empty" piece. During MODE_PLAY, only the first three are randomly picked, and the latter is used only in the idle state to tell other modules like *test_pos_rot.v* to not to fill up the elements of *placed_tetrominos[]* while the game clock is running. This module also enumerates 9 different tasks to enact user input such as rotations by updating the current coordinates as long as they don't run into the board or cause an intersection. The *remove_row* task also serves to keep track of which row to remove, and how to increment the seven segment adder. It is triggered by the

remove_row_en signal which means that all elements in one “flattened” row from the *placed_tetrominos* array has been filled.

Task	Purpose
move_left	Move curr_x left if !intersection & curr_x>0
move_right	Move curr_x right if !intersection & curr_x>0
rotate	Rotate within bounds and if !intersection
add_to_placed_tetrominos	Update array with the 4 current blocks
get_new_block	Reset the drop timer, pick a random piece, set curr_x and curr_y to the initial drop position, and set *game_clk_rst* to give user leeway
move_down	If within boundaries, incr curr_y, o/w call *get_new_block* & *add_to_placed_tetrominos*
drop_to_bottom	Switch to MODE_DROP
remove_row	Update score & the row that needs clearing
start_game	Switch to MODE_PLAY

Table 2. The Nine Tasks Helping the FSM Switch States and to Receive User Input

There are three smaller modules that we decided not to dedicate a section to because of the simplicity of their function. We will explain the following here:

Module	Purpose
game_clock.v	Create a 1 Hz clock, resets if *game_clk_rst*
debouncer.v	Debounce each button, and for switches have an enable and disabled signal to keep track of the transition from high to low and vice versa.
randomizer.v	Output a random 2 bit digit excluding 0 based on the clock

Table 3. Three Helper Modules and their Functions

2.2 Calc_cur_blk.v

The inputs of this module is the piece type, the rotation, and the x-y coordinates in terms of blocks. The output are four 8-bit values of each of the piece's blocks' indices within the *placed_tetrominos* array, as well as the height and width values, both ranging from 1-4. The purpose of this module is to convert the x-y block coordinates into indices of the flattened array based on rotation and piece type, and using the aforementioned equation 1. These four indices are represented by the signals *blk_1-4* and will be used to corroborate with the *test_pos_rot.v* module whether or not user input will cause a boundaries conflict with the board, and not to ensure that no two pieces overlap. Notably, the O piece does not depend on rotation, but the T piece has 4 different encodings to represent its position.
Assume *pos_x* = 3, and *pos_y* = 4, rot = 0 and BLOCK_WIDTH = 10.
If piece = O-piece:

Blk_1 = 43, Blk_2 = 44, Blk_1 = 53, Blk_1 = 54

If piece = I-piece:

Blk_1 = 43, Blk_2 =53, Blk_1 = 63, Blk_1 = 73 (straightened)

If piece = T-piece:

Blk_1 = 44, Blk_2 = 53, Blk_1 = 54, Blk_1 = 55 (inverted T position)

2.3 Test_pos_rot.v

The inputs to this function include the enable signals for each of the buttons, as well as the same inputs to *calc_cur_blk.v* module. The outputs are *test_pos_x, test_pos_y,* and *test_rot*, which contain the "test" x-y coordinates and rotation state based on the user input. The whole purpose of this module is to perform the calculation of the current x and y coordinates, as well as the rotation based on which button signal is high. For example if btn_left_en is set to high, then *test_pos_x = curr_x - 1* to represent that the block has shifted left within our array representation. Using the *intersects_placed_tetrominos* function in the *simple_tetris.v*, we can check these test coordinates do not intersect in combination with the current heights and widths.

2.4 Score_display.v

The inputs of this module is to take four 4-bit registers representing each digits and output the *an* and *seg* signals controlling the seven segment display. Using a small four state FSM, we cycle through the four digits and use the *display_digit* function to encode the *seg* output, using a 250 Hz display clock.

2.5 Complete_row.v

The inputs to this module include a boolean and the *placed_tetrominos* array, while the outputs include a boolean, *remove_row_en* and the row value that needs clearing, *remove_row_y*. The former output checks performs an AND operation on all the elements of a row in the flattened array, with an output of 1 meaning that all elements in it are occupied. In addition, the row is returned to the main module to switch mode to MODE_CLEAR. In this state, the FSM will be able to perform the *remove_row*() function and increment the score.

2.6 Vga640x480.v

This module was implemented using the same parameters from the NERP demo, as well as a similar method for cycling through the current pixel's x and y coordinates, *hc* and *vc*. The inputs included the current block positions in the flattened array *cur_blk_1-4*, the *cur_tetromino,* and the *placed_tetrominos* array. In order to keep track of the current blocks, we keep a register, *cur_blk_index* that converts the *hc* and *vc* coordinates into the current index of *placed_tetrominos*. Essentially the same encoding using equation 1 is used to convert the current pixel's coordinates into the index, where hbp and vbp represent the horizontal and vertical backporch, which are our board's starting points.

"A" = PIXELS_PER_BLOCK
cur_blk_index = ((*hc*-hbp)/A) + (((*vc*-vbp)/A)*BLOCK_WIDTH);

This register allows us to loop through *hc* and *vc* as was done in the NERP demo to update the board colors based off *cur_tetromino* and whether or not each board piece has been occupied, through *placed_tetrominos*. In addition the outside of the board will remain black, while the board itself will be orange, and each piece its own color. This operation was done by checking if *cur_blk_index* equalled any of the inputs *cur_blk_1-4*.

3. Simulation

To ensure that the *calc_cur_blk* module was outputting the right indices, we ran a simulation with the three different pieces and the four possible rotations to ensure all four blocks were properly calculated. *Rot* represents the rotation 0 meaning no rotation and every increase a +90 degree rotation, and *piece* represents piece type, from I, O, to T in that order.

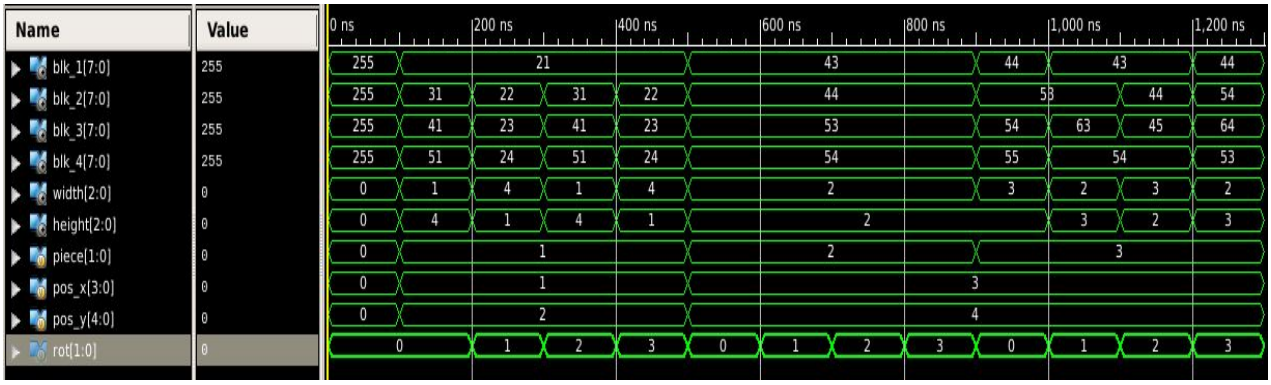

Figure 2. The Simulation output displaying the four block indices

After doing our own calculations, we confirmed that each rotation on each piece outputted the correct index for all four blocks. This can be seen in the way that the I piece retains 2 pairs of identical indices, because rotation value 0 is equivalent to rotation value 2, as is 1 to 3. Similarly, the O piece retains the same indices despite rotations.

The next module we needed to verify output the correct result was the *test_pos_rot mod* module. In this module we simply needed to check whether the *test_pos_y* would increment by 1 every clock cycle and also listen to user input.

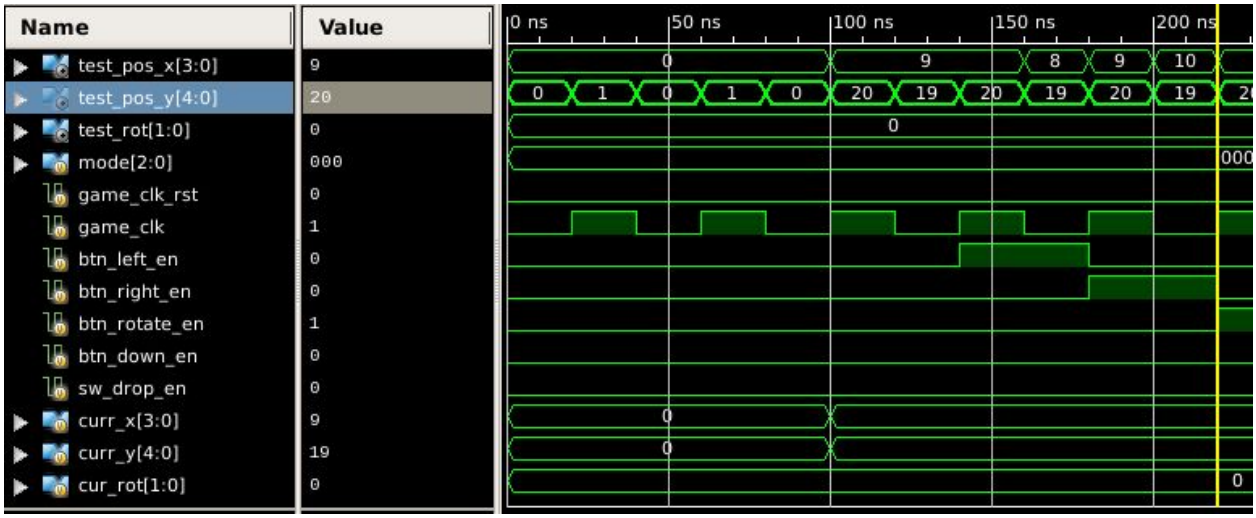

Figure 3. The Simulation output displaying test coordinates and rotation

The testing showed that user input such as *btn_right_en* and *btn_left_en* triggered correct the correct effect on *test_pos_x*, and that *test_pos_y* was being decremented by 1 every clock cycle. During real calls, the inputs to this function would also be changing with the same clock, so we wouldn't see each output value returning to previous states as can be seen above.

Regardless, the inputs trigger the correct change in the test coordinates every cycle, which is what we are looking for.

Lastly, we needed to check that the segment display FSM was outputting the right *an* and *seg* signals. By changing the input values of each digit we just needed to make sure that the right digit was being selected, and that the right segments were lit.

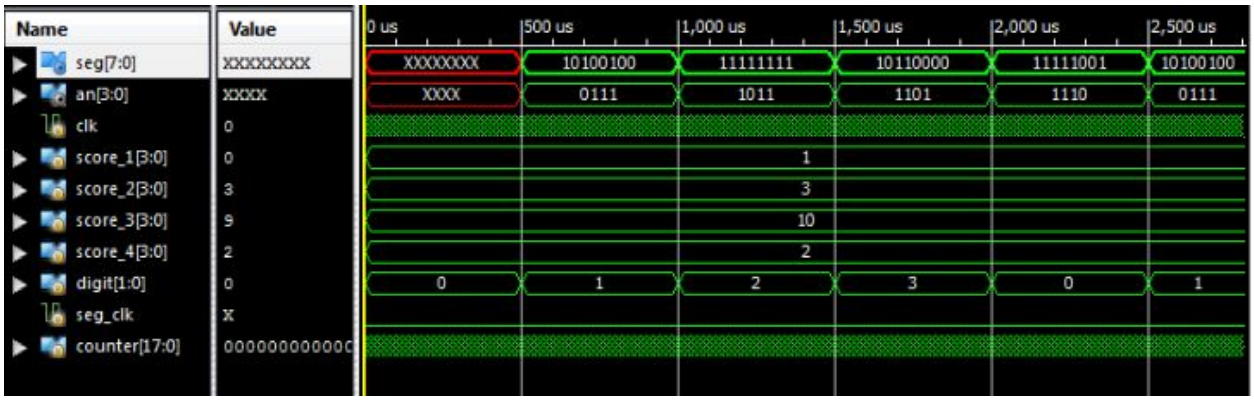

Figure 4. The Simulation output of Score_display_tb.v

As seen above, digit 0 corresponds to the rightmost digit, and each digit value past that cycles through. The *seg* signal can be matched to 8-bit value representing 1, 3, 10, and 2 as provided in previous lab manuals.

We chose not simulate the VGA module simply because it was easier to look at the physical screen for clues in how to debug our program. Unfortunately, we could not simulate the highest level module due to an unexpected crash every time ISIM was opened. The message included 4 instances of Error:801 “Unsupported variable” despite it having compiled in the ISE window. Regardless, the .bit file compiled and ran smoothly on the Nexys board.

4. Conclusion

In conclusion, our implementation of the simple tetris game consisted of a main top module that was supported by several lower level modules, each implementing a specific function required for the game to run such as the vga display module, row clearing module, block position calculating module, etc., as detailed above. Following this design, we managed to successfully implement the core game functionalities of our initial proposal. However, given the short time that was allotted to us for the project, there are still small bugs within our design, such as how the game will not start sometimes if the game is reset while it is still paused and then un-paused after. Another bug is that the drop mode switch may sometimes drop pieces when the switch is turned off as well as when it is turned on. Much of the difficulty we faced in this

project lied in getting the game to correctly display using the vga module. We had planned to have the game board display at the center of the screen, but due to time constraints, we decided to simply have the board display on the very left side instead. Another plan we had included having the color of the block stay the same color even after placement, but since this was not necessary for our game, we did not implement it. Other difficulties included figuring out how to calculate block locations for each rotation for each specific block, randomizing which piece would appear, and getting the seven segment display to work correctly. But other than these small bugs and difficulties, the game looks, works, and plays as intended.