

ECE 385

Fall 2018

Final Project

Tetris

Angela Park - aspark4@illinois.edu
Mihir Kumar - mkumar12@illinois.edu
Section: AB7 (Friday 11:00AM-1:50PM)
TA: Hadi Asghari-Moghaddam,
Vikram Anjur

Introduction

The purpose of our final project was to create the game Tetris using pure hardware logic in SystemVerilog. To implement our design in SystemVerilog, we had to connect a VGA monitor to display graphics, as well as a PS/2 keyboard to control the movement of the Tetris blocks. The graphics for our game were designed in Piskel and were integrated into our project through frame buffer and on-chip memory.

Written Description

Memory

For our Tetris game, we decided to use on-chip memory to store our sprites. We chose on-chip memory because we knew from lecture that there are about 3.888 Mbits of available on-chip memory, and we knew our Tetris design would not be graphically intense and would not require a lot of memory. We also planned on using a color palette for drawing our sprites, which would further reduce the amount of memory needed to store the sprites. Furthermore, we knew on-chip memory operates at a very high speed and is easier to initialize, as it can be done directly in SystemVerilog. Both these reasons factored into the decision to use on-chip memory rather than SRAM or SDRAM. Based on our compilation report, our entire Tetris implementation only used about 2% of the total available memory bits of on-chip memory (see below).

Total memory bits	65,852 / 3,981,312 (2 %)
-------------------	----------------------------

Sprites

To make sprites, we used the Piskel website (<https://www.piskelapp.com>) to create individual sprites for each Tetris block shape and their orientations. We also created sprites for the Tetris game logo, the numbers for keeping score, as well as the word 'SCORE'. We then used *png_to_palette_relative_resizer.py* from Rishi's Helper Tools to convert the .png files into .txt files. Each line in a .txt file contains a number that corresponds to a pixel's RGB hex code in our color palette, which is in our *colors* module. We decided to use a 10-color palette - black, cyan, yellow, red, orange, green, blue, purple, and two "transparent" colors (black board color and background gradient color). We decided to use a color palette when making the sprites in order to save memory space (see *Memory* section).

Because the sprites are drawn within rectangular boxes, the rectangle will contain pixels that make up the sprite's shape, as well as pixels outside of the sprite's shape. The pixels not part of the sprite's shape will be colored with the "transparent" color. Because there is no RGB hex code for "clear" or "transparent", what we really mean by "transparent" color is that we want the pixels outside of the sprite's shape to correspond to the color of the background that the sprite is overlaying (see *Figure 1*). If the sprite was a Tetris block shape, we would want the transparent color to correspond to the game board's background color, which is black. If we were drawing the Tetris game logo, the 'SCORE' word, or the score numbers, we wanted its transparent color to correspond to the game's background gradient color.



Figure 1. Shown above is a Tetris block shape sprite and the Tetris logo sprite. The pink areas in both sprites are pixels that we want to correspond to the "transparent" color.

In our *Color_Mapper* module, we accessed the data from each line of the .txt file for the block shape sprites by addressing the file using the following equation:

```
address = ((DrawX - X_Pos) + (DrawY - Y_Pos) * block_size_x);
```

Similarly, to obtain the data from the .txt files for the 'SCORE' word sprite, the number sprites, and the logo sprite, we had to use:

```
address=
((DrawX-(starting x position))+(DrawY-(starting y position))*width);
```

By using `$readmemh`, we were able to initialize memory from the .txt file and load the data into a memory array called OCM.

When creating the block shape sprites, we created sprites for the 7 different Tetris block types (I, O, T, S, Z, L, J) as well as their orientations for a total of 19 different block shape sprites (see *Figure 2* below). Each Tetris block shape consists of 4 "squares", where each square is 9x10 pixels (including the black outline around the square). The square dimensions were chosen to be 9x10 pixels, since they allowed us to create Tetris block shapes large enough to play with, but small enough that it was practical to create a game board around the sizing convention. Also, proactively breaking up each

sprite into constituent squares meant that we could easily map sprite data to an array using indexing and integer division. This ability greatly helped us in performing logical checks on the sprites for rotation, linear motion of sprites, and storing and retrieving relevant data from the board, which we also set as an array. This will be described in more detail in the sections below.

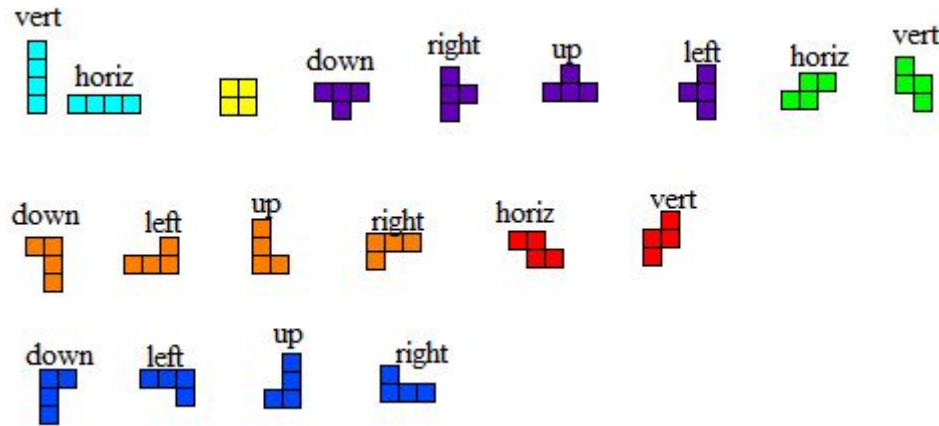


Figure 2. Shows all block types and their orientations. There are a total of 19 different block sprites.

We made sprites for numbers 0-9 in order to display the player's score on the VGA monitor, which is displayed underneath the 'SCORE' word sprite on the screen. To display the player's score on the monitor, we first made a register called *score*, which is incremented once every time a line is cleared in the *board* module. This value is then output to the *Color_Mapper* module for additional logic to convert the score into sprites that can be viewed on the monitor. This is done by breaking the score into its tens and ones digit. We did not need more digits, because we decided to cap the score at decimal 99 as a design decision. It encompasses all ranges of scores our game would need to practically process. The following equations let us easily find the two digits:

```
score_ones_digit_value = (score % 10)
score_tens_digit_value = (score / 10)
```

We then use a case statement with *score_ones_digit_value* as a select signal that corresponds to a number from 0-9. Depending on the value of the select signal, the case statement will choose the corresponding number to draw as the ones digit of the score. Similarly a case statement for *score_tens_digit_value* as a select signal will draw the corresponding tens digit number of the score.

Game Dimensions

Once we were able to get the required sprites to load onto the on-chip memory and display properly on the VGA monitor, we proceeded to decide upon the game board grid size and location. We decided the size of the grid to be 11 squares for the X dimension and 22 squares for the Y dimension for aesthetic reasons. In the original NES Nintendo Tetris game, we knew the size of the grid was 10x20 squares, but we wanted our grid to be a little larger to allow for more playing space while keeping a 1:2 ratio.

In our code, the bounds of the game board were defined by specifying the starting and ending X and Y coordinates of the game board and using conditional statements in the module *Color_Mapper* to paint a grid over this area to distinguish this area from the rest of the screen to the player. We added code to print a grid overlay on our game board window to improve the aesthetics of the game and to allow us to visualize the kind of data structures and algorithms that would be best suited to easily manage the game state. Since the square size of each sprite is 9x10 pixels, including the block border, the following equation allowed us to decide where to draw the grid:

```
((DrawX % (square_X_dimension)) == 0) ||
((DrawY % (square_Y_dimension - 1)) == 0))

=> ((DrawX % 9) == 0) || ((DrawY % 9) == 0))
```

DrawX: X coordinate of the current pixel being drawn by VGA

DrawY: Y coordinate of the current pixel being drawn by VGA

square_X_dimension: Length of each sprite square in X dimension

square_Y_dimension: Length of each sprite square in Y dimension

We subtract 1 from the Y dimension since there is a 1 pixel wide border on both ends of the square in Y dimension while the border exists on only one end of the square in X dimension (see *Figure 3*).

We subtract square_X_dimension and square_Y_dimension by 1 before using it for the modulus operation since the grid itself occupies 1 pixel of width both in X and Y dimensions. This pixel would also correspond to the 1 pixel border of the sprite square. Hence we're not losing any color data by covering colored portions of the sprite with the pixel, keeping our sprite storage efficient.

We set `grid_on` to high whenever this condition equates to True or 1'b1 and decided to paint the current pixel in grey color indicating it as part of the grid.

This resulted us in successfully getting the grid to appear over the specified game board area. Since we only want to make the grid appear over the game board, we added to following conditions that need to be true before setting `grid_on` to high.

```
((DrawX >= 270) && (DrawX <= 369)) &&
((DrawY >= 135) && (DrawY <= 333)))
```

This set of conditions constrain the grid logic to operate within the bounds of the game window which spans from pixel coordinates 270 to 369 in X dimension and pixel coordinates 135 to 333 in Y dimension as also mentioned before. The logic described so far in this section is implemented in *Color_Mapper* module.

Our next objective was to align the newly generated Tetris block with the grid overlay. This required some experimentation and we finally got the intended results upon using 297 and 135 as the starting X and Y coordinates of all newly generated blocks respectively.

The game board location was chosen to be from pixels 270 to 369 in the X dimension (100 pixel width) and from pixels 135 to 333 in the Y dimension (199 pixel height) since these dimensions result in a centered game window both along X and Y axes. The 100 pixel width by 199 pixel height then corresponded with the 11 by 22 square grid we wanted for our game, if each square was 9x10 pixels of data. Our sprite data fits this requirement since it consists of 8x8 squares of color content with a 1 pixel border on all sides of the square, which is also overlapped by the grid (see *Figure 3*). Hence, each square in the sprites needs to be 9x10 pixels in size to appear in alignment with the grid.

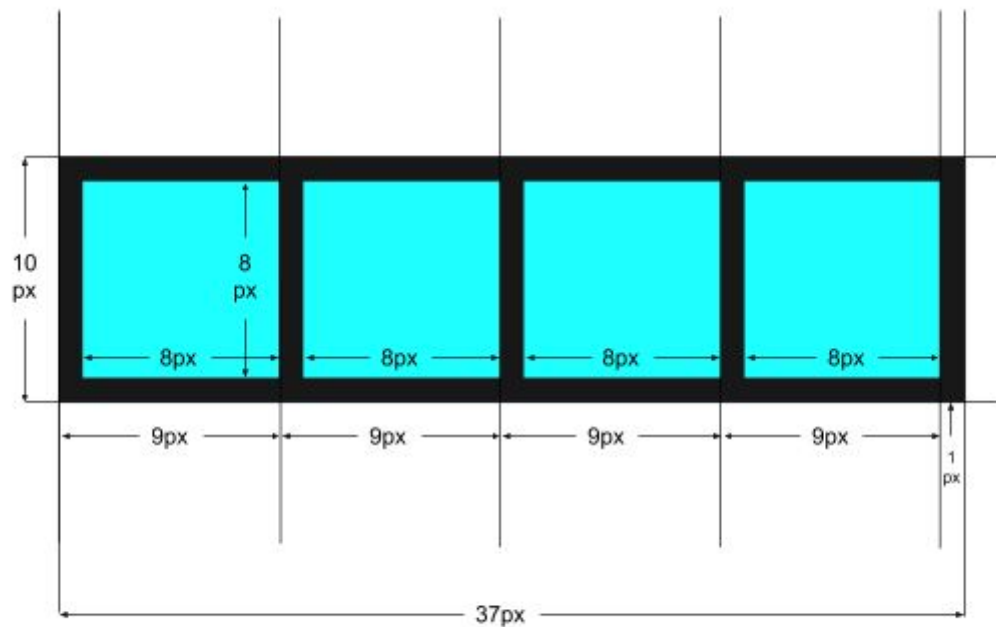


Figure 3. Illustration of pixel sizes of constituent squares in sprites
(Horizontal I block sprite shown as an example)

The repeated “square” of each sprite is 9x10 pixels in size with 8x8 pixels of color and 1 pixel of border on three sides. The rightmost pixel is an extra pixel of border not contained in a repeatable “square”. The same design applies to the Y dimension as well. Given this design choice, the game window/grid size can be calculated using the following equations:

$$X_Max = X_Min + 9 * (\text{number of blocks desired in X dimension}) + 1$$

$$Y_Max = Y_Min + 9 * (\text{number of blocks desired in Y dimension}) + 1$$

Since we decided to have 11 blocks in the X dimension, 22 blocks in the Y dimension, X_Min at 270 pixels, and Y_Min at 135 pixels from the top-left of the monitor, our dimensions come out to be:

$$X_Max = 270 + 9 * 11 + 1 = 370$$

$$Y_Max = 135 + 9 * 22 + 1 = 334$$

Note that we need to reduce 1 from each of these values since arrays start indexing at 0, which makes:

$$X_Max = 370 - 1 = 369$$

$$Y_Max = 334 - 1 = 333$$

These calculations led us to set our game window from 270 to 369 pixels in the X dimension and 135 to 333 in the Y dimension.

Block Motion

The linear motion of blocks is performed using a technique similar to the one used for moving the ball in lab 8. The key difference from the method in lab 8 is that instead of letting one keypress let the ball move continuously in one direction until it bounced off an edge, we added logic to move the block in the left or right direction by the step size only once.

We wanted the Tetris block to move only one step size per keypress, rather than letting the block move continuously. To do this, we added a signal called *move_horiz*, which is set low by default, and will go high *only after* the block is directed to move in the horizontal direction. The conditions that allow the block to move is when the proper key (A or D) is pressed, if the block is within the board boundaries, and if the signal *move_horiz* is set low. Therefore, by setting *move_horiz* high *after* these conditions are met, we can prevent the block from moving continuously in a direction after a single keypress or if the key was being held down. We then used a counter to control when we update the block's x-direction motion. The counter increments by 1 every rising edge of the frame clock when *move_horiz* is high. The x-motion of the block is then updated when *move_horiz* is high and when the counter reaches 6. This allows granular control of the block's movement which is of key importance to the gaming experience of Tetris.

For the block's movement in the y-direction, we used another counter to control the speed at which the block fell. The counter increments by 1 on every rising edge of the frame clock. Once the counter reaches 50, the movement of the block in the y-direction is updated and the counter is reset to 0. Therefore, the block will move downwards by the step size once every 50 clock cycles.

When the S key is held down, we wanted the block to speed up in the y-direction. To implement this feature, we added a condition in which if the key was being held down, a signal called *fast_drop* would go high. If this signal was high and if the counter reached 5, rather than 50, the block's y-direction movement would be updated and the counter set back to 0. Therefore, the block will move downwards by the step size once every 5 clock cycles rather than once every 50 clock cycles.

The rotational motion of blocks is controlled using a state machine consisting of 8 states (see *Figure 4*). The state machine will start in the initial state, *Rotate_0*, which sets the rotate signal, *rot_sel*, to 1. The *rot_sel* signal, along with the *block_sel* signal in the *choose_block* module, will determine the type of block and its orientation, and that information will then determine which block shape sprite to draw. Upon a keypress for rotating the block (W), the state machine will check whether the current position of the block and the size of the next rotated shape will go past the set boundaries of the game board. If so, the state machine will remain in its current state and will not allow the block to rotate. If the position and the next rotated shape's size is within the boundaries, the state machine moves on to a wait state that waits for the keypress to go low (waits for the key to be released). This prevents multiple rotations upon a single keypress (see *Bugs* section below). Once the keypress is low, the state machine goes on to the next rotate state and sets a different *rot_sel* signal. This new *rot_sel* signal will correspond to the block's rotated shape. By going through these rotate and wait states, the shape can be rotated clockwise upon multiple keypresses.

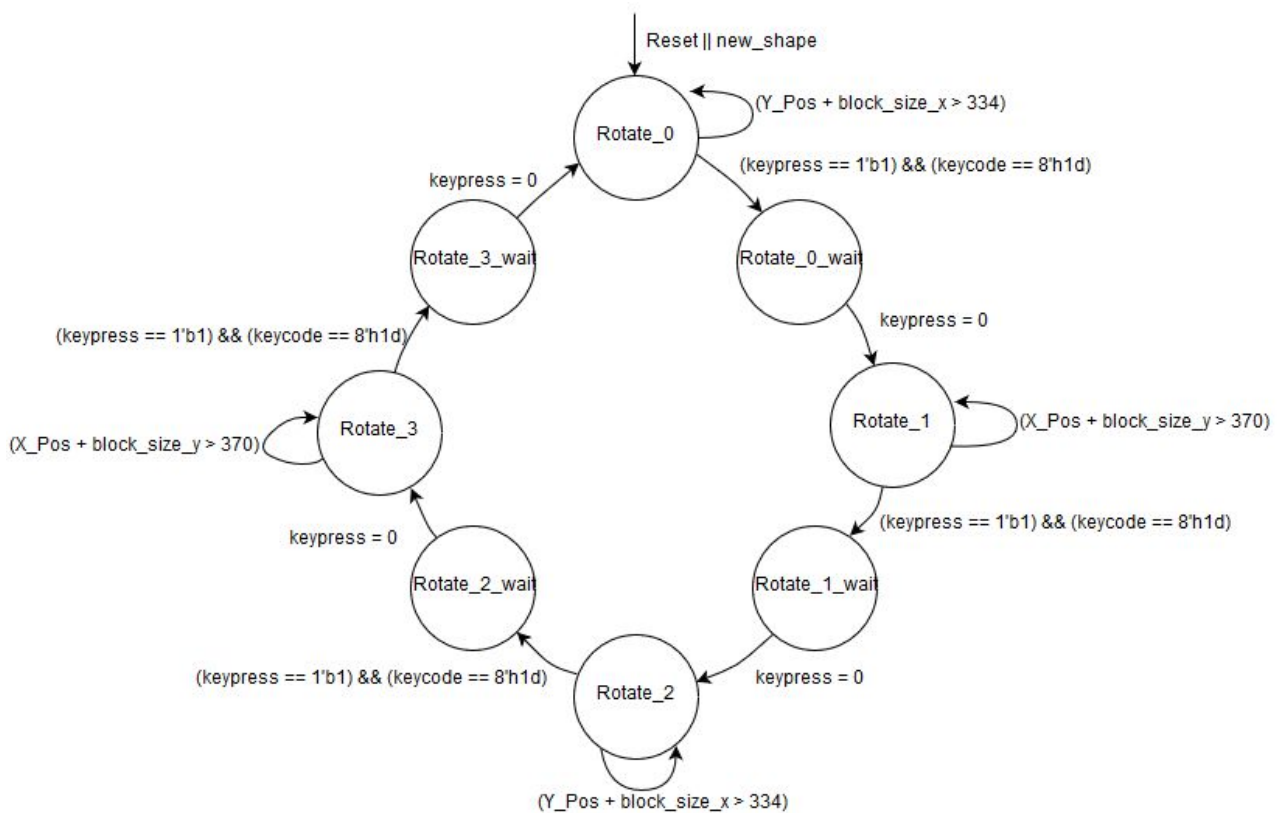


Figure 4. Shows state diagram for the state machine that controls the block's rotation in the *rotate_block* module.

W (ROTATE)	0x1D
S (FAST DROP)	0x1B
A (LEFT)	0x1C
D (RIGHT)	0x23

Figure 5. Shows scancodes for PS/2 keyboard key presses used in this lab, and their corresponding action.

Board

The final step was to architect a design to synchronize the active block with the past state of the game at any time. There are two underlying steps in achieving this synchronization:

The first step was to retrieve information about the immediate surroundings of the active block within its span of rotation and the block's adjacent grid cells to detect possible collisions with the edge of the game window and other blocks saved in the past state of the game. If we detect a possible collision, we simply do not allow the movement to occur or save the block's position to the state and generate a new block. Validating the movement before it occurs also ensures that the block will never occupy illegal/occupied/out-of-bounds space.

The second step was to save the state of the board when the block either reaches the bottom of the window or stacks on top of a pre-existing block. To perform this action at the correct time, we also need to check the immediate vicinity of the block.

Our design choices so far have led us to generate blocks divided into constituent squares of the same size (9x10 pixels) and a grid that visually represents the game state as a 2D grid of these squares. We have also managed to correctly implement linear movement and rotation of blocks in such a space. It is important to realize that the common underlying unit of design here is the 9x10 square pixel, each of which only occupies a single color at a time. Storing the board as a 2D array of size 11 columns by 22 rows, corresponding to the number of squares in each dimension in the grid would result in an easy to use structure where we could simply use the current position of the block, its type, size, and orientation to find any information about its surroundings that we might need. Hence, we store the board as a 2D array where each element of the array holds a 4-bit index that corresponds to a color (as defined in the module *colors*). To use such a structure to hold the

board, we would also need to map each pixel location in the game window to a specific index of the board. Such a mapping between every pixel coordinate in the game window and its corresponding index can be performed using the following equation employing integer division:

```
pixel_drawX_drawY =
board[((y_coordinate-Y_Min)/9)][((y_coordinate-X_Min)/9)]
```

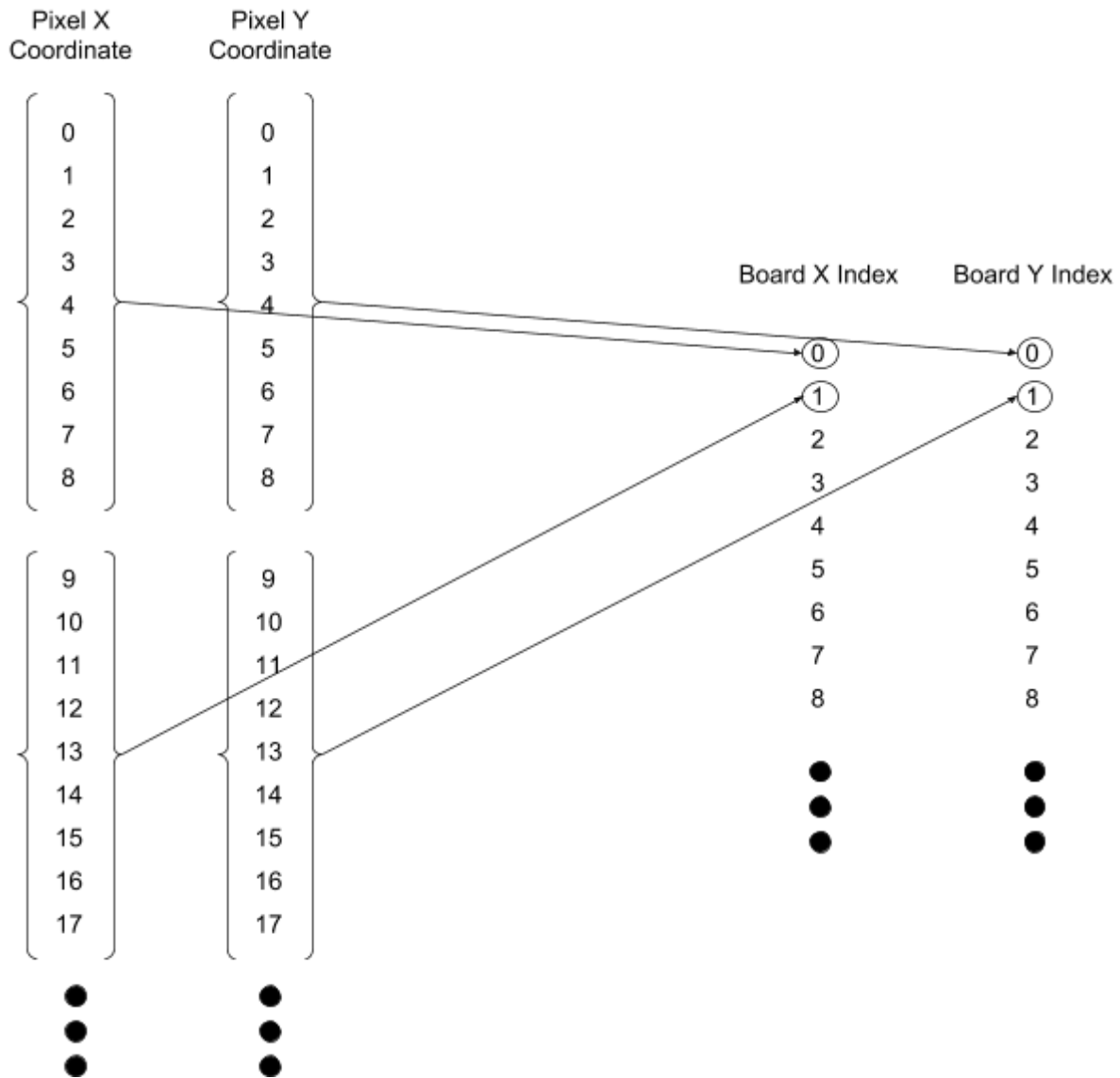


Figure 5. Shows mapping between sets of pixel values and their corresponding board indices.

Figure 5 demonstrates this mapping via integer division. The triple dot symbol is used to indicate further increasing values in the sets and board indices.

The location of the current block is stored as `X_Pos` and `Y_Pos`. They refer to the pixel coordinates of the top-left pixel of the block. Using these signals and the translation between pixel coordinates and board indices, we can find all the information required in the first step of synchronization, using basic 2D array indexing. This approach to storing the game state also allows us to easily index and set the required board array elements to the color of the current block depending on its shape and orientation, completing the functionality of the game window.

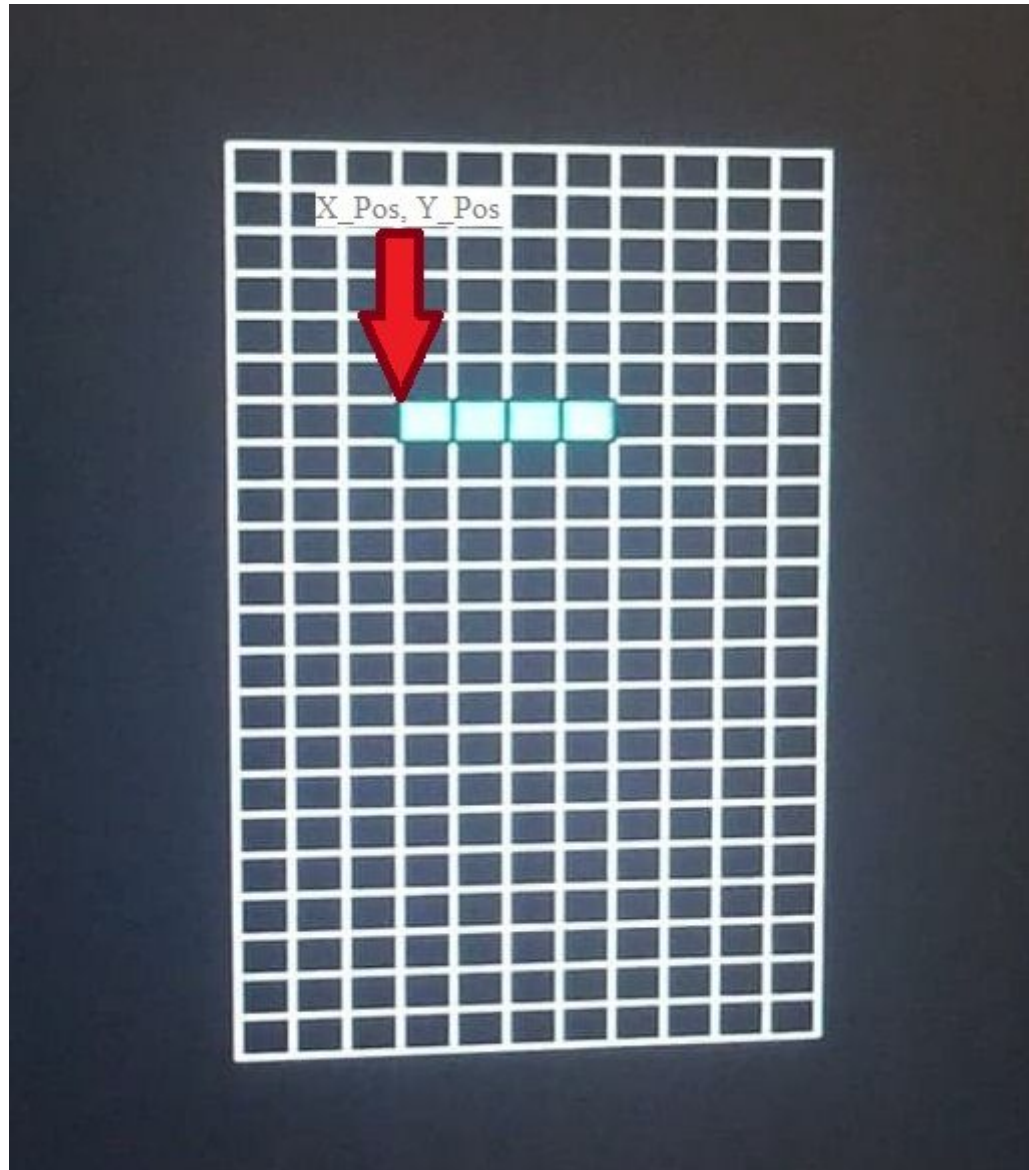


Figure 6. Shows the pixels that correspond to the current block's location in our design.

Relevant Diagrams

Block diagram

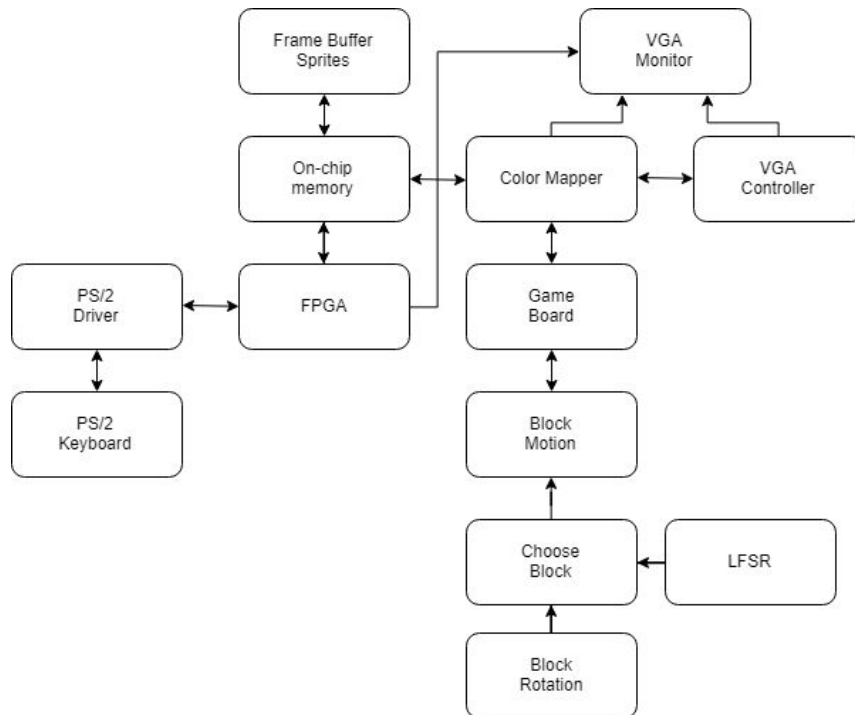


Figure 7. Shows the block diagram for our Tetris game design.

State diagram

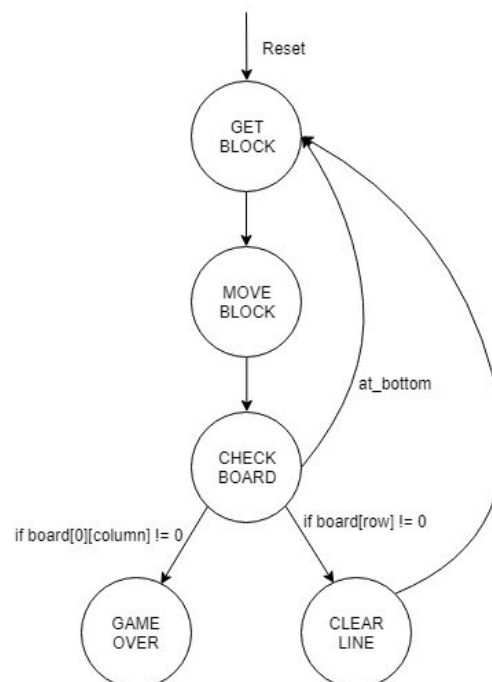


Figure 8. Shows state diagram of our Tetris design. Game will begin when Reset is triggered. Random block is chosen and will move down on game board. Board is checked and will either clear a line if a full row of the board is full, and then get another block. If any column of row 0 (top row of board) is filled, game is over. If neither clearing line or ending game, *at_bottom* signal is triggered and state will get new block.

Module Descriptions

Module: tetris_top.sv

Inputs: CLOCK_50, [3:0] KEY, PS2_CLK, PS2_DAT

Outputs: [7:0] VGA_R, [7:0] VGA_G, [7:0] VGA_B, VGA_CLK, VGA_SYNC_N, VGA_BLANK_N, VGA_VS, VGA_HS, [6:0] HEX0, [6:0] HEX1, [6:0] HEX2, [6:0] HEX3, [6:0] HEX4, [6:0] HEX5

Description: This is the top-level module of the design.

Purpose: The purpose of this module is to connect all internal modules together. This module also outputs the proper VGA signals to the monitor, and outputs the score, the block_sel, and the rot_sel to the hex displays.

Module: sprite_ram.sv

Inputs: Ck, [18:0] addr

Outputs: [3:0] data

Description: This file contains all sprite ram modules for each sprite used in the project.

Purpose: These ram modules initialize memory from a .txt file. The data, which corresponds to a value in the *colors* module, is loaded into a memory array based on an address determined in the *Color_Mapper* module.

Module: Color_Mapper.sv

Inputs: [9:0] DrawX, [9:0] DrawY, [3:0] tetris_block_data, [3:0] score_data, [3:0]

O_block_data, [3:0] I_vert_block_data, [3:0] I_horiz_block_data, [3:0]

Z_vert_block_data, [3:0] Z_horiz_block_data, [3:0] S_vert_block_data, [3:0]

S_horiz_block_data, [3:0] J_up_block_data, [3:0] J_right_block_data, [3:0]

J_left_block_data, [3:0] J_down_block_data, [3:0] L_up_block_data, [3:0]

L_right_block_data, [3:0] L_left_block_data, [3:0] L_down_block_data, [3:0]

T_up_block_data, [3:0] T_right_block_data, [3:0] T_left_block_data, [3:0]

T_down_block_data, [3:0] score_0_digit_data, [3:0] score_1_digit_data, [3:0]

score_2_digit_data, [3:0] score_3_digit_data, [3:0] score_4_digit_data, [3:0]

score_5_digit_data, [3:0] score_6_digit_data, [3:0] score_7_digit_data, [3:0]

score_8_digit_data, [3:0] score_9_digit_data, choose_O, choose_I_horiz,

choose_I_vert, choose_Z_horiz, choose_Z_vert, choose_S_horiz,

choose_S_vert, choose_J_left, choose_J_up, choose_J_right, choose_J_down,

choose_L_left, choose_L_up, choose_L_right, choose_L_down, choose_T_left,

choose_T_up, choose_T_right, choose_T_down, [9:0] X_Pos, [9:0] Y_Pos, [5:0]

block_size_x, [5:0] block_size_y, [21:0][10:0][3:0] board, [15:0] score

Outputs: [18:0] address, [7:0] VGA_R, [7:0] VGA_G, [7:0] VGA_B

Description: This module obtains the RGB values for each pixel. The address for the sprite ram modules are also determined in this module.

Purpose: This module outputs an address to the *sprite_ram* module to obtain the data from its corresponding memory address. Depending on the choose signal from the *choose_block* module, the corresponding data will then be input into the *colors* module instantiated within this module in order to assign the RGB colors to the corresponding pixels. These RGB values are then output to the toplevel. Also handles the logic for displaying the score and grid as an overlay.

Module: colors.sv

Inputs: [3:0] decided_color, [9:0] DrawX

Outputs: [7:0] Red, [7:0] Green, [7:0] Blue

Description: This module is a 10-to-1 MUX.

Purpose: Depending on the select signal, this module assigns the RGB hex codes and outputs them to the *Color_Mapper* module.

Module: choose_block.sv

Inputs: at_bottom, Clk, Reset, [7:0] LFSR_out, [1:0] rot_sel

Outputs: new_shape, [2:0] block_sel, choose_O, choose_I_horiz, choose_I_vert, choose_Z_horiz, choose_Z_vert, choose_S_horiz, choose_S_vert, choose_J_left, choose_J_up, choose_J_right, choose_J_down, choose_L_left, choose_L_up, choose_L_right, choose_L_down, choose_T_left, choose_T_up, choose_T_right, choose_T_down, [5:0] block_size_x, [5:0] block_size_y

Description: This module handles generating and selecting the size and orientation of the new block.

Purpose: Decides whether a new block should be generated when the Reset button was pressed or if the current block reached the bottom of the game board/landed on top of another block. It selects the type of the newly generated block depending on LFSR_out and sets its X and Y dimensions as well as its orientation based on the rot_sel signal from the *rotate_block* module.

Module: LFSR.sv

Inputs: Clk, Reset

Outputs: [7:0] LFSR_out

Description: This module is an external 8-bit LFSR. Creates pseudo-random number generator for tetris block generation.

Purpose: Loads seed into module upon hitting the Reset button. A feedback signal is generated by taking bit 4 and bit 2 of LFSR_out and XORing them. LFSR_out is left shifted, in which the MSB is thrown out and the feedback signal becomes the new LSB of the LFSR_out.

Module: rotate_block.sv

Inputs: Clk, Reset, keypress, [7:0] keycode, new_shape, [9:0] X_Pos, [9:0] Y_Pos, [5:0] block_size_x, [5:0] block_size_y

Outputs: [2:0] rot_sel

Description: This module contains a state machine that determines the orientation of the block.

Purpose: The state machine in this module contains four rotate states and four corresponding wait states, for a total of eight states. There are four rotate states because there are four maximum possible orientations per shape. With each press and release of the rotate key (W), the state will go through one rotate state and one wait state. The state machine will start in an initial rotate state with a rotate select signal that corresponds to a particular orientation, and upon the keypress of the rotate key will move to the wait state. The wait state moves to the next rotate state upon the release of the key, in which a new rotate select signal will be given. This module also checks that the block's next rotate shape will be constrained within the boundaries of the game board, otherwise it will not allow the block to rotate.

Module: block_motion.sv

Inputs: Clk, Reset, frame_clk, [9:0] DrawX, [9:0] DrawY, [7:0] keycode, keypress, [5:0] block_size_x, [5:0] block_size_y, choose_O, choose_L_horiz, choose_L_vert, choose_Z_horiz, choose_Z_vert, choose_S_horiz, choose_S_vert, choose_J_left, choose_J_up, choose_J_right, choose_J_down, choose_L_left, choose_L_up, choose_L_right, choose_L_down, choose_T_left, choose_T_up, choose_T_right, choose_T_down, [21:0][10:0][3:0] board

Outputs: at_bottom, [9:0] X_Pos, [9:0] Y_Pos

Description: This module sets game parameters and handles linear movement of the block in X and Y directions.

Purpose: Sets the X and Y coordinates of the new block, size of the game window and step size of the block. Resets the block coordinates and motion logic when the Reset button is pressed or when the current block reaches the bottom of the game board/lands on top of another block. Contains logic to prevent overlapping of active block with blocks already placed on the board. This is done by checking the position, orientation and type of the block being moved, and then by checking relevant adjacent grid cells. The module will check whether the block will run into another block in the X direction, and if so, will prevent the block from overlapping with the detected block. Also contains logic to check if block will hit the bottom of the board or another block in the Y direction. If so, signal *at_bottom* is set to high, which will trigger a new block to generate at the top of the game board. Depending on the key pressed, (A) or (D), it will move the current block in the left or right direction respectively. On pressing (S) it will make the block drop in the Y direction much faster,

similar to most Tetris implementations. This module also contains logic to keep the current block within game window boundaries during movement.

Module: board.sv

Inputs: Clk, Reset, at_bottom, choose_O, choose_I_horiz, choose_I_vert, choose_Z_horiz, choose_Z_vert, choose_S_horiz, choose_S_vert, choose_J_left, choose_J_up, choose_J_right, choose_J_down, choose_L_left, choose_L_up, choose_L_right, choose_L_down, choose_T_left, choose_T_up, choose_T_right, choose_T_down, [9:0] X_Pos, [9:0] Y_Pos

Outputs: [15:0] score, [21:0][10:0][3:0] board

Description: This module contains logic for initialization and manipulation of the array representing the game board when the current block needs to be saved to the game board.

Purpose: Assigns color value of the current block to certain game board cells depending on the orientation, shape and position of the block when it needs to be saved. Also checks if the any row of the board, from bottom to top, is filled with non-empty / black color values and clears the row(s) where this condition is met, moving each subsequent row down by the number of rows that are cleared. The player score is incremented by 1 upon each row getting cleared. The player score and updated board array are output to *color_mapper* for further processing before being printed to the monitor.

Module: VGA_controller.sv

Inputs: Clk, Reset, VGA_CLK,

Outputs: VGA_HS, VGA_VS, VGA_BLANK_N, VGA_SYNC_N, [9:0] DrawX, [9:0] DrawY

Description: This module determines the horizontal and vertical sync timing signals.

Purpose: This module determines the horizontal and vertical sync timing signals needed for VGA monitors. The electron beam scans horizontally from top to bottom and “paints” each pixel by keeping track of the coordinates.

Module: vga_clk.v

Inputs: inclk0

Outputs: c0

Description: This acts as a 25 MHz pixel clock signal.

Purpose: This module acts as the clock signal for our VGA controller module. It is generated by PLL.

Module: keyboard.sv

Inputs: Clk, psClk, psData, reset

Outputs: [7:0] keyCode, press

Description: This module handles interactions with the keyboard over a PS/2 port.

Purpose: This module contains a driver for the PS/2 keyboard which we use to interact with the game. It detects when a key is pressed and handles the incoming scancodes by passing it to instances of *reg_11* modules for further processing.

Design Resources and Statistics

LUT	44,849
DSP	(see Figure 2 below)
Memory (BRAM)	119,808
Flip-Flop	1150
Frequency	24.13 MHz
Static Power	107.63 mW
Dynamic Power	346.29 mW
Total Power	576.1 mW

Figure 9. Shows design statistics table.


Fitter DSP Block Usage Summary				
 <<Filter>>				
	Statistic	Number Used	Available per Block	Maximum Available
1	Simple Multipliers (9-bit)	0	2	532
2	Simple Multipliers (18-bit)	1	1	266
3	Embedded Multiplier Blocks	1	--	266
4	Embedded Multiplier 9-bit elements	2	2	532
5	Signed Embedded Multipliers	0	--	--
6	Unsigned Embedded Multipliers	1	--	--
7	Mixed Sign Embedded Multipliers	0	--	--
8	Variable Sign Embedded Multipliers	0	--	--
9	Dedicated Input Shift Register Chains	0	--	--

Figure 10. Shows DSP block usage summary.

Bugs

Initially, we could not get the LFSR to work because of the way we implemented it. We originally thought that it was because of the way we were trying to implement the actual LFSR module. We later realized it was because we set `block_sel` to `LFSR_out`, and because `LFSR_out` is constantly changing and generating random numbers, the `block_sel` was therefore also constantly changing. This resulted in us seeing the block shape very rapidly changing and glitching on the screen. To fix this problem, we created a signal in the *choose_block* module called *new_shape*, which was set high whenever a Reset was detected or if the *at_bottom* signal, which indicated whether a block reached the bottom of the board/landed on another block, was detected. The `block_sel` signal would only be updated when *new_shape* was set high, and then *new_shape* would be set low until another Reset signal or *at_bottom* signal was detected.

When trying to get the block to move in the x-direction, the block would move in the specified direction upon a single keypress without stopping (like the ball in Lab 8). We wanted our block to move only once step size per keypress instead of moving in a direction continuously upon the keypress. To fix this problem, we added a signal called *move_horiz* (see *Block Motion* section).

The state machine for getting the select signal, *rot_sel*, for rotation wasn't working initially. The select signal would go straight from 0 to 3 (checked using the hex displays) when it should have been going from 0 to 1 to 2 to 3. This was because multiple rotations would rapidly happen upon a single keypress. We needed to add wait states in between the actual rotate states that would wait for the key to be released before moving on to the next rotate state.

We also ran into a bug in which if rotated the Tetris shape near the borders of the game board, the rotated shape would go outside of the borders. In order to fix this, we added additional constraints in the *rotate_block* module (see *Block Motion* section).

In our design, we overlay the current block on top of the board to correctly print it. We also treat each sprite as a solid rectangle with areas within the sprite shape filled with color and areas outside the sprite shape marked with a "transparent" color (see *Sprites* section for explanation on this). These transparent areas were colored as black for the block shape sprites to blend in with the black background of the game board. As a consequence of this set of design choices, we had a bug where these "transparent" areas of the current block would paint over the board, making colored components of the board look black during the block's movement. To fix this, we had to add a condition that checks if a square in the current block is empty, the color at that pixel is set to the color of the board at this pixel. This fixed the bug universally for all sprites.

Conclusion

In conclusion, we were able to get the Tetris game working after fixing the above mentioned bugs. This final project gave us an opportunity to use the skills we learnt in the class so far and apply them to bring an idea of our liking to life. It gave us flexibility in creating the game in many different ways. We had the option to create the game using the SoC or use only hardware. We chose to make the game in pure hardware to challenge ourselves and to apply our design ideas with what we learned about SystemVerilog in this class. Furthermore, we knew hardware would be a lot faster than if we were to also use software in our design.

References

During our research phase, we found documentation regarding the original Nintendo Tetris game developed for NES:

<https://hackaday.com/2014/01/28/a-deep-dive-into-nes-tetris/>
http://meatfighter.com/nintendotetrisai/#Table_of_Contents

According to the link (go to section *Picking Tetriminos* in linked webpage), Nintendo used a 16-bit Fibonacci linear feedback shift register (LFSR) as a pseudorandom number generator (PRNG) to randomly generate their Tetris blocks. We wanted to do something similar to Nintendo's LFSR method, so we found example codes implementing an 8-bit LFSR here:

<https://www.quora.com/What-would-be-the-verilog-code-for-8-bit-linear-feedback-shift-register>

We used this code to make an 8-bit LFSR for our random Tetris block shape generator (see *LFSR.sv* in *Module Descriptions* section).

We also used the [PS/2 Keyboard Driver](https://wiki.illinois.edu/wiki/display/ece385fa18/Final+Project) by Sai Ma and Marie Liu linked on the ECE 385 final project page (<https://wiki.illinois.edu/wiki/display/ece385fa18/Final+Project>) to implement the PS/2 keyboard into our Tetris game design.