ECE 385

FALL 2023

Final Report

Final Report

Meredith Naylor, Jacob Poeschel

Friday 2:45 PM

Nick Lu

Introduction:

The purpose of this project was to create a functional Tetris game exclusively in SystemVerilog. To achieve this, we combined many different modules. First, we instantiated our necessary sprites in ROM files to make access easier. Additionally, we imported the block diagram, vga_controller, and personal IP to access user input through the keyboard like we did in lab 6. Then, we created an FSM to control the overall flow of the program and determine when to show certain screens, when the blocks should begin falling, when score should count, and when new blocks should be generated. Block generation and movement is handled in our blocks module. This module initializes each of the seven shapes Tetris blocks may hold at any given time. Each of these individual block modules contain programming to allow for rotation and movement while checking for collisions and remaining in bounds. The top level blocks module selects which block position variable from the seven shapes to use based on the current value of our sprite select. Sprite select is determined by a pseudo random cycle through roughly fifty different shapes which transition on a counter. All of this information is then passed to our color mapper, fpb_1_example.sv. This file uses the current block position and its current state to draw it on the grid. At the beginning of each cycle, the existing blocks are also drawn. These blocks are added to our board registers representing the grid in the top level file where we have access to the signal representing when a block is placed. The top level also handles row deletion and scoring. Each part of this functionality is described in more detail below.

List of Features:

- Start screen with Tetris logo
- Game Screen which includes:
    - Tetris logo
    - Game board
    - Score
- Falling tetris blocks
- Using keys to rotate the tetris block
- Using keys to move the tetris block
- Collision detections for all blocks and bounds
- Clearing a row once the row is completely full
- Prompting game over when a block reaches the top row
- Incrementing score based on block placements and row deletion
- Displaying a variable score
- End screen with game over message
- Using a key to reset the tetris game

Functionality:

Sprite Storage:

In order to create a realistic Tetris game, we needed access to twelve total sprites. These include two background screens, the tetris logo, press any key and game over text bubbles, and

the seven sprites necessary to draw each Tetris shape. Each of these sprites are stored in ROM and are drawn using palettes which were derived from the helper tools given. The helper script allowed us to specify a desired image resolution and also a specific number of color bits per pixel. In general, we made all of our images as low resolution as possible to preserve FPGA space. We created a block ROM for each sprite specifying the width, depth, and the COE initialization file provided to us by the helper tool. Additionally, we imported the provided palettes. Each of these ROM blocks and palettes are instantiated in the color mapper files where the sprites are drawn. Then, to access each sprite, we simply used DrawX and DrawY to calculate the address which should be read from ROM and assigned the returned data into the output colors. This is the complete set up which is used to draw all of our sprites for our project.

Finite State Machine:

The FSM controls all state transitions and logic for our Tetris game. The game begins in a start state, where it is instructed to show the start screen sprite. This state also serves as a reset state since it will always be entered before the game begins. So, if the flag to show the starting screen is high, game variables such as score, multiplier, and the game over are reset to their starting values. Additionally, the board memory which contains the current blocks on the grid is cleared to prepare for a new game. Pressing any key triggers the transition into the generate block state where the first block is generated. This state raises a generate block signal which is passed into the tetris_blocks module and is used to begin drawing a block at the top middle of the screen. It is critical that this state is separate from the standard game state, as combining the two would cause the block to constantly reset to the top of the screen and no block would drop properly. Once the block is generated, the FSM enters the game state. It will remain in this state

until it receives a high game over signal. This signal is generated in the top level when the top row of the board is checked. If any block is present, the game is over. This initiates a transition into the final state, the game over state. This state begins the drawing of the end screen and prepares for a reset into the start state. Overall, these states allow us to control the flow of our game logic.

Board Management:

To store the board we created a set of registers board_regs in the top level. Board_regs is 6 bits wide with a depth of 200 bits. We use the top bit, bit [5], to determine if a block is placed in that position of the board or not. The next two bits, bits [4:3], are used to determine the rotation of the block. The bottom three bits, bits [2:0], are used to determine which block is at that location in memory. We use the board_regs to check collisions on rotation and moving logic, check if a row should be cleared, and check for game end. This set of registers is used in two places; each tetris block module and the top level file. In the logic for each tetris block, we must use the registers to determine collisions. In the top level, we must determine the current game state based on the blocks stored in memory. These behaviors are described in detail in the following sections.

Collisions:

Each tetris block must be able to move left, right, down, and rotate. In order to achieve each behavior, a unique set of conditions must be validated by checking the current state of the board. Our board registers contain one register for each grid location, making it relatively simple to index into them to check for collisions. At the beginning of the logic for each block, the

current grid index is calculated based on the current BlockX and BlockY. These X and Y values represent the pixel location for the very top left of each sprite, allowing them to be scaled by the block_size to create the row major order index of the grid. Then, since each grid location is represented by a single register, checking the grid location to the left or right of the current index is achieved by a shift of -1 and +1 respectively and checking locations above or below is achieved with a shift of -10 and +10 respectively. Despite the relative simplicity of these calculations, they become bulky due to the scale of their use. Each of the seven sprites must perform on average three collision checks for all of the possible movements left, right, down, and rotate. Additionally, these sprites each have a different set of four movements for each of the four possible block states. Aside from checking for collisions with other blocks, we also must ensure that all movements are in bounds. Although these calculations were very tedious, they allow us to guarantee that every possible move for a falling block is valid.

Row Clearing:

Row clearing is handled in the top level file along with block placement and the determination of game over. Since our registers are indexed in row major order, a full row is determined by a set of ten consecutive indices having a high fifth bit (the fifth bit signifies if the block is full or empty). This means that a for loop beginning at index 199 and being incremented by -10 every cycle sets up the skeleton for each row to be checked. For each index, we must also check each of the 9 indices before the current index, or in other words, we need another for loop to loop between index and index - 9. This set of 10 values represents the corresponding indices for each row. If at any point, board_regs[idx][5] is equal to 0, we may break the loop and move on to the next row. However, if the loop finishes, every block in the row is full, and that row must be cleared. Consequently, all blocks in the following rows must be shifted down to account

for the space left by the cleared row. To shift the blocks down we loop through all the movable indices backwards. If a block right below the current space then we do not move the current block down. If there is not a block below the current space then we move the current block down one space. At the end of the down movement we clear the top row. Then after all blocks have moved we check to see if there are any blocks in the top row. If that condition is true then the game is over.

Game Over:

The game ends when any block enters the top row of the grid. To determine this, we must check the board stored in memory for any full blocks in the top row. If a full block is encountered, a game over signal is raised and subsequently sent to the FSM to begin the transition into the end state.

Block Placement:

When checking for movement in each individual block module, if downward movement is blocked, the block placed signal is raised. This signal, along with the current block state, are output into the top level. Based on the output state, board_regs are filled with the location of the block that was just placed. Each block orientation fills out a different set of registers which allow for the placed shape to be maintained in board memory.

Tetris Block Logic:

Because every tetris block has different conditions for movement, each of them need their own individual module to control their movement. Despite the differences in collision check indices, each block module follows the same general format. Additionally, using macros to represent basic block attributes like width and height allow the code to be generalized and repeated without having to change individual details for each block. First, if generate block is high, and it has chosen the current block, the block's position is reset to the top middle of the board. Gravity is implemented on a counter to ensure that despite using the 25 MHz pixel clock, each block only shifts down once per second. To do this, collisions must be checked for every index directly below the position of the current block. If there is a collision with another block or with the bottom of the grid, the current block position remains the same and the block is marked as placed. This triggers the sequence described above to write the block into memory. After implementing gravity, the other movement directions are checked on a counter which activates 6 times per second which allows for more fluid movement than the movement due to gravity. The movement direction is determined by the input key press which is held in a buffer. This ensures that a key can only be pressed a maximum of six times a second while simultaneously guaranteeing that a key press which does not perfectly coincide with the ⅙ of a second counter is not lost by the time the counter is reached. Depending on the current block state and the desired movement direction, the corresponding collision checks are made using the board_regs which are passed into every block module. If the move is valid, the block's position is updated. If it is invalid, the block's position is simply maintained. Once movement has been handled, rotation is checked. Aside from the standard collision checks which closely resemble those necessary for movement, in some cases, the block's position may need to be updated to shift diagonally or in

any direction to represent some of the more unconventional rotations. To simplify collision checks and storage in memory, the block's position is strictly set to the top left corner of the sprite, even if the top left grid position may actually be empty in the case of certain sprites (ie L shape top right cell). Once again, this generalized the process of implementing the block logic. Cumulatively, this code handles block position and permits all desired movement and rotation. However, one final key element is block selection in the top level module. Based on the current value of the sprite select bits, we choose which of the individual module's positions to set to our main position and update to the top level. Additionally, we also must select which block placed signal we care about. Other than that, the top level tetris_blocks module implements each individual model. Overall, the combination of this top level module and the individual modules allow us to control block movement.

Sprite Selection:

Sprite selection is done in the sprite_select module. When generate block is raised, a sprite is selected based on the value of a counter which cycles through a pseudo random set of blocks. We have ensured that each sequence of blocks is a valid sequence so that our game matches the tetris standard. The selected sprite value is output on two separate signals, sprite_select and long_sprite_select. This is necessary as sprite_select gets reset to 0 every time generate block returns to 0. This allows us to have a signal to reset a given blocks position to the top of the grid only once while not interfering with block movement any time the block should not be reset. The long_sprite_select variable is never reset, so it maintains the value representing the current sprite throughout the entire duration of its existence. These 7 bit values are one hot encoded to represent the set of seven blocks.

Scoring:

Scoring is handled entirely in the top level. It is a 20 bit value allowing for a maximum score of 999999. It is assigned a base multiplier and also a base increment, which can be changed to modify how each game is scored. Currently, score increments every time a block is placed and also increments when a row is cleared. Since we receive the block placed signal and also clear rows in the top level, it is trivial to increment the score in these places. The challenging part of managing a score is dynamically changing what is essentially an integer number into a series of six digits drawn on the screen. This is achieved through the use of a modified font_rom which contains the ten digits 0-9. Every clock cycle, each digit of the score integer is translated into a bit sequence containing the value of each digit which can be used to index into font rom. The current digit is calculated by taking score % 10, and stored in a digit variable. Then, score is shifted to the right by a factor of 10 by taking score / 10. To guarantee that every score is translated into a six digit number, this math occurs in a for loop beginning at 0 and ending at 5. For each digit, its value is assigned as a four bit sequence into the digit_bits array. The for loop index is used to determine where each digit is stored. By setting digit_bits[i *4 +: 4], we set the first digit (ones place) into the least significant bits of digit_bits, and then build the array out from there. This means that looking from left to right in digit_bits shows the desired number in binary, making drawing each digit significantly easier. This completes the manipulation of the score and leaves drawing to be implemented based on digit_bits in our color mapper file.

Drawing:

Our drawing logic can be generalized into four sections: background, grid, blocks, and scoreboard. Each of these sections are described in detail below. Generally, our implementation

involves drawing from two separate color mapper files fpb_example and fp_start_example. Fp_start_example draws the start and end screens while fpb_example draws the game screen. Each of these files instantiate the roms and palettes for any sprites they use. Another critical aspect of our implementation is utilizing a sequence of if statements to draw. Essentially, we check if we are in a location, and if we are, we draw the desired sprite. This allows us to determine the exact X and Y values to begin drawing the sprite. Importantly, we must always adjust the drawX and drawY values to be 0 in the top left corner of the sprite so that all addressing is in bounds. Another major issue determining a method to choose which pixels in a sprite to draw or ignore. Some sprites are not clean rectangular shapes and must only be drawn within specific bounds. For all such sprites, the pixels which should not be drawn were colored in pure green (RGB 0, 15, 0) in Microsoft paint. Then, when we are in bounds to draw a sprite, we only actually draw the sprite if the color is not bright green. Choosing bright green as our exclusion color worked very well with our warm theme that does not naturally contain very much green. With this setup, drawing logic is described below.

Background Sprites:

Background sprites include the background images as well as sprites such as the Tetris logo or game over message. One key element of these sprites is that they are intended to be drawn behind everything else, meaning they are drawn last. In fp_start_example, we draw one of two screens depending on whether the game is starting or ending. For both, the same background image is used. One critical idea is that the majority of our sprites are very small when compared with the screen (usually smaller than 120x80). This means that in order for a background image to fill the screen, its addressing must be manipulated so that several consecutive pixels all draw from the same address, effectively stretching the image. Similar address manipulation is used

when accessing virtually every background sprite, so to avoid redundancy it is only described here. Now that we have drawn the background image, both screens also contain the Tetris logo which is scaled to take a more prominent position on the screen. Finally, the press any key or game over sprites are also drawn.

Grid:

Since the grid is made up of 16x16 squares with the boundary lines being white and the inside being black, we decided to save ROM space by manually drawing the grid ourselves. By shifting X to equal 0 at the beginning of the grid, if we draw white every time X % 16 = 0 or Y % 16 = 0 and black otherwise, we will have the basic grid layout. This is also preferred over having the grid as a sprite because the white grid boundaries should be drawn above the blocks while the black grid background should be drawn behind them. If these were one sprite, it would be challenging to achieve this separation. Overall, drawing the grid manually was simple and effective.

Blocks:

In our color mapper file, we have access to both the block position and its current state. Once again, despite it being a tedious process, setting boundaries for each possible state for each block is simple thanks to our knowledge of the position and the state. For blocks which are being drawn from memory, calculating the address is also easy. When we are reading from the board_regs, we are checking one block at a time. This means that for the current cell, we just have to draw any 16x16 part from the sprite. In most cases, since the top left position of the sprite is a full block, we can just shift X and Y to equal 0 at the top left of the grid cell, and address directly into memory. For some blocks where the top left of the sprite is not a block but

rather empty space, we just need to shift our X address by 16 in the X direction. On the other hand, drawing the dropping block requires more math. Since we must take the current rotational state of the block into consideration to correctly adjust shading, calculating the row major order based on the standard X and Y does not always suffice. Our sprites are stored in memory in their vertical states. So, for any block rotated 90 or 270 degrees, instead of DrawX representing the current column and DrawY representing the current row, they instead flip so that DrawX represents the current row and DrawY represents the current column. This means that we must calculate different equations for each state to ensure that they determine the correct sprite address. Additionally, we must check to ensure that the color at the selected address is not bright green to allow for the necessary pixels which correspond to blank space in the sprite to be ignored. By combining these methods for drawing blocks from memory and drawing the dropping block, we are successfully able to model the entire Tetris board.

Scoreboard:

Drawing the scoreboard begins with laying out a box for the characters. Since we have a six digit score and each character is eight pixels wide and sixteen pixels tall, we must allow at least 48 pixels horizontally and 16 pixels vertically. To allow for some extra space, this area is expanded to a 64 by 24 area. In the top level file, score was translated into a 24 bit signal called digit_bits which contains six 4 bit digits representing the digits of the current score. Going from left to right, we need to draw these digits in reverse order (from bit 24-1). By shifting the current X to the start of the score characters and dividing it by 16, we can calculate which character we are currently on. Then, we must account for the reversed order of the characters, so we subtract that character from five. This way, if X = 4, we calculate 4/8 = 0 and 5 - 0 = 5 demonstrating that we must fetch the information for the fifth character. We access the desired character through a

case statement. In this case, the fifth character would point us to digit_bits[23:20]. Now, we have the bits representing the current digit. The ten digits are stored in order in ROM with 16 addresses for every character. Since we have the value of the digit we must draw, we can shift to that character by multiplying that value by the standard address calculation involving drawX and drawY. Similarly to how we drew this font rom in lab 7, we then subtract 7 from the current pixel to draw accounting for the backwards order of accessing the ROM. With this implemented, we are able to draw a variable score on the screen.

Software Connection:

The software components of this lab are all of the given functions combined with the four we created which allow for communication between the MicroBlaze and the USB keyboard through SPI. To achieve this communication, we used the SPI transfer function to read and write bytes of data from the MAX3421E and into the top level SystemVerilog file. All of these functions must select and deselect the desired slave device at the beginning and end of the function, respectively. To do this, we use the XSpi_SetSlaveSelect function. To set the slave, we use the member variable SlaveSelectMask associated with the given SpiInstance. Since the mask selects via a one hot encoding style, to deselect we can simply pass in a mask that equals 0. Additionally, each of these functions requires instantiation of write and read buffers which are responsible for handling the data. Write buffers must be filled out with the given values and passed into the transfer function. Read buffers must remain empty and will be filled in with values by the transfer function. It is also critical to remember that the first byte in both buffers is reserved for the command byte, meaning each buffer must be nbytes plus 1 length. And, when we are looking for the data read into the read buffer, we must begin with the data at index 1.

Another important aspect of every function is checking the return code to make sure that the transfer is completed successfully. The specifics of each function are described below.

MAXreg_wr:

This function is responsible for writing a single byte. To signify a write, the write buffer is loaded with the desired register value in binary plus 10 which acts as the write enable signal. In this case, we do not need to read any data, so the read buffer can be passed as NULL. The write buffer is filled with the given value making its total length two bytes. This function does not return anything.

MAXreg_rd:

This function is responsible for reading a single byte. To signify a read, the write buffer is loaded with just the register value, leaving the second bit 0 to represent a read. It is critical to understand that even for a read, a write buffer must be passed which contains the command byte. Therefore, our buffer length is once again two bytes. The value read from the transfer is stored in the read buffer's first index. This function returns the read value at index 1.

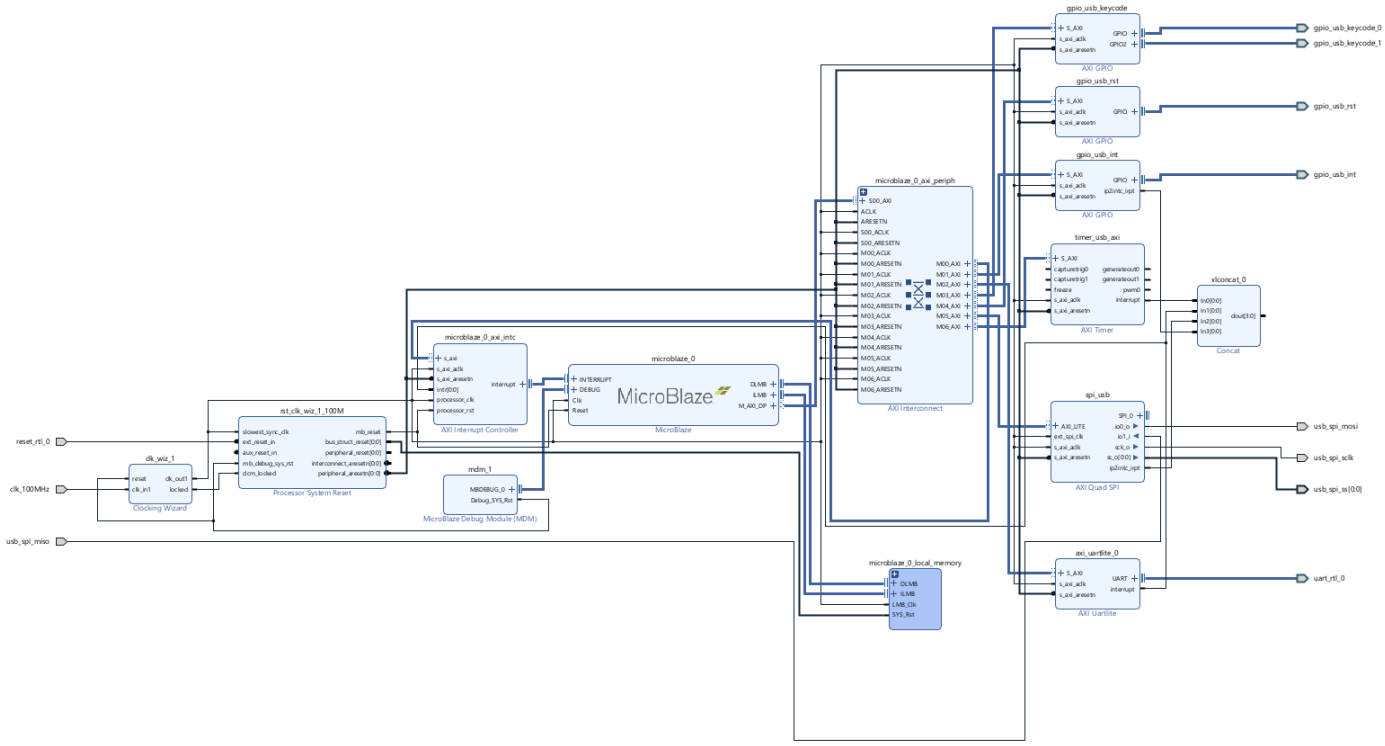
MAXbytes_wr:

This function writes in a similar way as the plain write function, except for the nbytes of data we must transmit. Still, the write buffer must be loaded with the command byte. Once again, the read buffer can be NULL. Then, the bytes from data are loaded into the write buffer, making it nbytes + 1 in length. Next, the transfer is completed. Here, we return data + nbytes to verify that the right information was written.

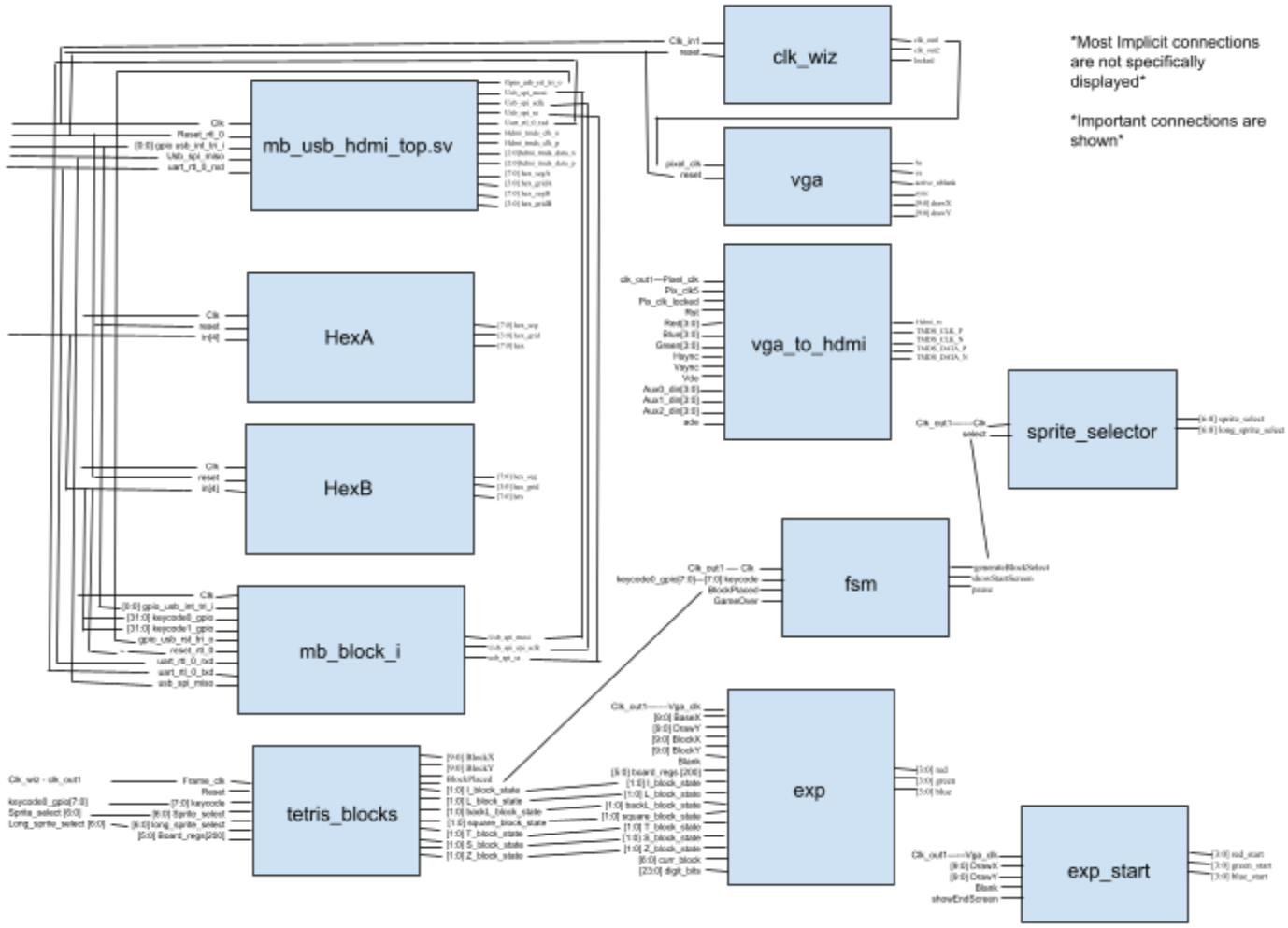
MAXbytes_rd:

This function is similar to the standard read function but it once again reads nbytes of data rather than just one. The write buffer must be filled with the command byte as usual. Then, the transfer is completed. Then, the values from the read buffer starting at index 1 are copied into the data array starting at index 0 that is given to us. We skip the first byte of the read buffer because it contains the command byte, which we do not want to copy over. Finally, we return the data array plus nbytes.

Block Diagram:



Top Level Design:



Module Descriptions:

Module: mb_usb_hdmi_top.sv

Input: Clk, reset_rtl_0, [0:0] gpio_usb_int_tri_i, usb_spi_miso, uart_rtl_0_rxd

Output: gpio_usb_rst_tri_o, usb_spi_mosi, usb_spi_sclk, usb_spi_ss, uart_rtl_0_txd, hdmi_tmds_clk_n, hdmi_tmds_clk_p, [2:0]hdmi_tmds_data_n, [2:0]hdmi_tmds_data_p,

[7:0] hex_segA, [3:0] hex_gridA, [7:0] hex_segB, [3:0] hex_gridB

Description: This is the top level file which contains all of our logic for this project. It instantiates all other modules and also controls the board_regs memory.

Purpose: Instantiate all other modules. Manage and update score. Provide functionality for reset. Write to board_regs if a block is placed. Perform game logic like clearing rows and checking for game over.

Module: HexDriver

Input: clk, reset, in[4], [3:0] nibble

Output: [7:0] hex_seg, [3:0] hex_grid, [7:0] hex

Description: The HexDriver.sv file is responsible for translating binary into hex output that can be displayed on the FPGA board. This makes it so that our input and output register can be displayed on the FPGA board. Because of this, we can monitor the register visually and confirm circuit functionality.

Purpose: Display result and other values into hex displays.

Module: VGA_controller.sv

Input: pixel_clk, reset

Output: hs, vs, active_nblank, sync, [9:0] drawX, [9:0] drawY

Description: Determines drawX and drawY values which represent the current pixel being drawn on the screen. It also is responsible for hsync and vsync which ensures the synchronization of drawing.

Purpose: Control drawing of the ball and gradient on the screen

Module: vga_to_hdmi

Input: pix_clk, pix_clk5, pix_clk_locked, rst, red[3:0], blue[3:0], green[3:0], hsync, vsync, vde, aux0_din[3:0], aux1_din[3:0], aux2_din[3:0], ade

Output: hdmi_tx, TMDS_CLK_P, TMDS_CLK_N, TMDS_DATA_P, TMDS_DATA_N

Description: This module converts the vga values given by the vga controller into values that can be used by HDMI.

Purpose: Translate the VGA signal into HDMI so it can be displayed on the monitor.

Component: clk_wiz_1

Input: reset, clk_in1

Output: clk_out1, locked

Description: Provides a user interface to easily configure the required clocks for this project.

Purpose: Creates the various clock speeds required.

Component: rst_clk_wiz_1_100M

Input: slowest_sync_clk, ext_reset_in, aux_reset_in, mb_debug_sys_rst, dcm_locked

Output: mb_reset, bus_struct_reset[0:0], peripheral_reset[0:0], interconnect_aresetn[0:0], peripheral_aresetn[0:0]

Description: This clocking wiz is used to generate a clock frequency of 100MHz.

Purpose: The purpose is to generate a clock frequency of 100MHz.

Component: microblaze_0_axi_intc

Input: s_axi, s_axi_aclk, s_axi_aresetn, intr[0:0], processor_clk, processor_rst

Output: interrupt

Description: This component is used to handle, prioritize, and route interrupt signals going to the microblaze.

Purpose: The purpose of this component is to handle interrupts.

Component: mdm_1

Input: MBDEBUG_0

Output: Debug_SYS_Rst

Description: The mdm_1 component of the block diagram is to instantiate the debug purpose of the microblaze. It provides access to breakpoints, watchpoints, and program state and memory during execution.

Purpose: Used for debugging purposes.

Component: microblaze_0

Input: INTERRUPT, DEBUG, Clk, Reset

Output: DLMB, ILMB, M_AXI_DP

Description: This component represents the core functionality of the microblaze CPU. It allows us to write C software to control the behavior of the various system components. Additionally, it allows peripheral connectivity which enables us to connect all of the requisite components.

Purpose: CPU core which controls circuit behavior.

Component: microblaze_0_axi_periph

Input: S00_AXO, ACLK, ARESETN, S00_ACLK, S00_ARESETN, M00_ACLK, M00_ARESETN, M01_ACLK, M01_ARESETN, M02_ACLK, M02_ARESETN, M03_ACLK, M03_ARESETN, M04_ACLK, M04_ARESETN, M05_ACLK, M05_ARESETN, M06_ACLK, M06_ARESETN

Output: M00_AXI, M01_AXI, M02_AXI, M03_AXI, M04_AXI, M05_AXI, M06_AXI

Description: This module is an AXI peripheral configured to work with the microblaze CPU using AXI protocol.

Purpose: Provides the slave interface for the microblaze.

Component: microblaze_0_local_memory

Input: DLMB, ILMB, LMB_Clk, SYS_Rst

Output:

Description: Instantiate on chip memory which is used to store program data by the microblaze processor.

Purpose: Provides low latency memory access for use by the microblaze.

Component: axi_uartlite_0

Input: S_AXI, a_axi_aclk, s_axi_aresetn

Output: UART, interrupt

Description: Enables the FPGA to transmit data serially to other circuit components. Also provides interrupt support.

Purpose: Allows FPGA to transmit data and handle interrupts based on that data.

Component: gpio_usb_keycode

Input: S_AXI, s_axi_aclk, s_axi_aresetn

Output: GPIO, GPIO2

Description: Connects the keycode inputs into the system via USB for use by microblaze.

Purpose: Sends keycode inputs into system.

Component: gpio_usb_rst

Input: S_AXI, s_axi_aclk, s_axi_aresetn

Output: GPIO

Description: Connects keycode reset input into the system.

Purpose: Connects input signal into system.

Component: gpio_usb_int

Input: S_AXI, s_axi_aclk, s_axi_aresetn

Output: GPIO, ip2intc_irpt

Description: Connects usb interrupt signal into system.

Purpose: Connects an input into the system.

Component: timer_usb_axi

Input: S_AXI, capturetrig0, capturetrig1, freeze, s_axi_aclk, s_axi_aclk, s_axi_aresetn

Output: generateout0, generateout1, pwm0, interrupt

Description: Determines the timing on specific timing events and transmits signals via USB.

Purpose: Control circuit timing.

Component: spi_usb

Input: AXI_LITE, ext_spi_clk, s_axi_aclk, s_axi_aresetn

Output: SPI_0, io0_o, io1_i, sck_o, ss_o[0:0], ip2intc_irpt

Description: Provides means for SPI to communicate with USB.

Purpose: Configures microblaze to communicate with USB via SPI protocol.

Component: xlconcat_0

Input: In0[0:0], In1[0:0], In2[0:0], In3[0:0]

Output: dout[3:0]

Description: Concatenates 4 input bits into one output bit.

Module: fsm.sv

Input: Clk, [7:0] keycode, BlockPlaced, GameOver

Output: generateBlockSelect, showStartScreen, showEndScreen, pause

Description: Finite state machine which controls all state transitions and incorporates game logic

Purpose: Control current state based on the input game signals. Output critical control signals which other modules use to update functionality.

Module: fpb_1_example.sv

Input: vga_clk, [9:0] BaseX, [9:0] DrawY, [9:0] BlockX, [9:0] BlockY, blank, [5:0] board_regs[200], [1:0] I_block_state, [1:0] L_block_state, [1:0] backL_block_state, [1:0] square_block_state, [1:0] T_block_state, [1:0] S_block_state, [1:0] Z_block_state, [6:0] curr_block, [23:0] digit_bits

Output: [3:0] red, [3:0] green, [3:0] blue

Description: This module is a color mapper which handles the drawing of the game board and game state sprites.

Purpose: This module draws the background, grid, score, blocks from memory, and falling blocks. It uses a series of if statements to choose what to draw for any given pixel.

Module: fb_start_example

Input: vga_clk, [9:0] DrawX, [9:0] DrawY, blank, ShowEndScreen

Output: [3:0] red_start, [3:0] green_start, [3:0] blue_start

Description: Another color mapper which instantiates the necessary sprites and palettes for drawing the start and end screens.

Purpose: Draw the start and end screens.

Module: tetris_blocks.sv

Input: frame_clk, Reset, [7:0] keycode, [6:0] Sprite_Select, [6:0] Long_Sprite_Select, [5:0] Board_Regs[200]

Output: [9:0] BlockX, BlockY, BlockPlaced, [1:0] I_block_state, [1:0] L_block_state, [1:0] backL_block_state, [1:0] square_block_state, [1:0] T_block_state, [1:0] S_block_state, [1:0] Z_block_state

Description: This module instantiates all of the Tetris blocks.

Purpose: Instantiate all necessary Tetris blocks, and select which BlockX and BlockY should be used to draw in every given cycle.

Module: tetris_I

Input: frame_clk, [7:0] keycode, GenerateBlock, [9:0] BlockX, [9:0] BlockY, [5:0] board_regs[200], active

Output: BlockPlaced, [1:0] I_block_state

Description: A module to calculate the movement of the Tetris I block.

Purpose: Check for collisions to determine if movement is valid or if the block has been placed. Update block position based on movement direction or rotation.

Module: tetris_square

Input: frame_clk, [7:0] keycode, GenerateBlock, [9:0] BlockX, [9:0] BlockY, [5:0] board_regs[200], active

Output: BlockPlaced, [1:0] square_block_state

Description: A module to calculate the movement of the Tetris square block.

Purpose: Check for collisions to determine if movement is valid or if the block has been placed. Update block position based on movement direction or rotation.

Module: tetris_L

Input: frame_clk, reset, [7:0] keycode,, GenerateBlock, [9:0] BlockX, [9:0] BlockY, [5:0] board_regs[200]

Output: BlockPlaced, [1:0] L_block_state

Description: A module to calculate the movement of the Tetris L block.

Purpose: Check for collisions to determine if movement is valid or if the block has been placed. Update block position based on movement direction or rotation.

Module: tetris_S

Input: frame_clk, [7:0] keycode, GenerateBlock, [9:0] BlockX, [9:0] BlockY, [5:0] board_regs[200], active

Output: BlockPlaced, [1:0] S_block_state

Description: A module to calculate the movement of the Tetris S block.

Purpose: Check for collisions to determine if movement is valid or if the block has been placed. Update block position based on movement direction or rotation.

Module: tetris_T

Input: frame_clk, [7:0] keycode, GenerateBlock, [9:0] BlockX, [9:0] BlockY, [5:0] board_regs[200], active

Output: BlockPlaced, [1:0] T_block_state

Description: A module to calculate the movement of the Tetris T block.

Purpose: Check for collisions to determine if movement is valid or if the block has been placed. Update block position based on movement direction or rotation.

Module: tetris_Z

Input: frame_clk, [7:0] keycode, GenerateBlock, [9:0] BlockX, [9:0] BlockY, [5:0] board_regs[200], active

Output: BlockPlaced, [1:0] Z_block_state

Description: A module to calculate the movement of the Tetris Z block.

Purpose: Check for collisions to determine if movement is valid or if the block has been placed. Update block position based on movement direction or rotation.

Module: tetris_backL

Input: frame_clk, [7:0] keycode, GenerateBlock, [9:0] BlockX, [9:0] BlockY, [5:0] board_regs[200], active

Output: BlockPlaced, [1:0] backL_block_state

Description: A module to calculate the movement of the Tetris backL block.

Purpose: Check for collisions to determine if movement is valid or if the block has been placed. Update block position based on movement direction or rotation.

Module: sprite_selector

Input: Clk, select

Output: [6:0] sprite_select, [6:0] long_sprite_select

Description: Select a shape to draw when triggered, and cycle through the pseudo random set of blocks.

Purpose: This module selects the current block which should be drawn.

Module: mb_block_i.v

Input: Clk, [0:0] gpio_usb_int_tri_i, [31:0] keycode0_gpio, [31:0] keycode1_gpio, gpio_usb_rst_tri_o, reset_rtl_0, uart_rtl_0_rxd, uart_rtl_0_txd, usb_spi_miso,

Output: usb_spi_mosi, usb_spi_spi_sclk, usb_spi_ss

Description: Block wrapper which provides the necessary signals for the IPs in the block diagram.

Purpose: Provide critical signals to various IP blocks.

Design Resources and Statistics:

LUTs	30,314
DSP	10
Memory (BRAM)	20
Flip-Flops	4,347
Latches	0
Frequency (MHz)	328.6
Static Power (mW)	76
Dynamic Power (mW)	491
Total Power (mW)	565

Conclusions from Design Resources and Statistics:

This project is by far the most computationally intense project we have worked on this semester. Having to implement separate collision checks and bounds for moving and drawing the seven different blocks in the four different rotation states necessitated a massive amount of logic. Significantly, we relied on our board_regs extensively and in several different modules. Another aspect of our project that consumed a large amount of LUTs was our 200 index for loop to perform game updates. For loops are extremely computationally intensive in SystemVerilog. Overall, we nearly ran out of LUTs in our implementation of Tetris, and focusing on writing efficient code is something to keep in mind in the future.

Conclusion:

Functionality of Design:

Our design met all of our project goals set in our project proposal.

Design Extensions:

Although our design functions as a base Tetris remake, there is a lot of room for improvement. Overall, our code is not very efficient. If we could generalize the logic to calculate movement and rotations for each block we could potentially save thousands of LUTs. Additionally, if we could find a way to shrink the board storage in memory or potentially use

BRAM instead we could make the project much more efficient. Communication conflicts are also another prominent issue with our project. In many instances we were forced to use blocking communication inside of always_ff blocks which is not recommended and could certainly create errors or slowdowns in our implementation. Cycles of always_ff blocks were also responsible for drawing delays which created errors where our block drawings were displaying a clock cycle too late. If we could combine code more efficiently we could likely eliminate these delays.

Aside from code efficiency improvements and reorganization, we could also add a lot of functionality. If we could shrink our LUT usage, we could implement additional features such as multiplayer, theme selection, and AI by building off of our existing project. Since we already have the software connection to read keycodes from a keyboard, creating multiplayer is as simple as drawing a new screen with two grids and processing both keycodes simultaneously. Additionally, adding theme selection could be easily achieved with a few extra states in the FSM and some if statement checks when drawing. AI would be more challenging, but programming an order of keycodes logically is not impossible, although we would ideally code this logic in C.

Overall, we are proud of how our project turned out!