ECE 385
Spring 2023
Final Project

Tetris on FPGA

Tyler Jurczyk (tylerjj3), Elliot Dziemiela (edziem2)
Section WR
Wei Ren

Introduction

For our final project we decided to implement a version of the classic game Tetris on the fpga. Using the keyboard, we allow a player to move a piece In the vein of classic tetris: pressing A moves the piece left, pressing D moves the piece right, pressing S makes the piece fall faster, and pressing W rotates the piece. The game lasts 2 minutes and speeds up as time winds down. Once the timer reaches 0 or a piece lands at the top of the screen, the game ends and can be reset by pressing key 0 on the FPGA.There is also a scoring system whereby clearing a single row gives you 1 point, 2 rows together 5 points, 3 rows together 10 points, and 4 rows together 25 points. We've implemented this game entirely in SystemVerilog on the FPGA – there is nothing being run on a processor besides the drivers for keyboard input run on the NIOS 2 using on-chip memory. We use SDRAM to hold our video memory for the game.

Written Description of System

Our game works by keeping track of a logical 10x20 board maintaining the presence of filled or unfilled squares, and a visual 10x20 board used as vram to hold 200 RGB values for each "square" of the game board. With that 10x20 bit array maintaining the logic layout of the pieces fallen on the board, the only dynamic element at each frame of our game is the single piece controlled by the player. By outputting drawing and clearing coordinates of this single piece to the vram controller tetris.sv, the game's logic tells tetris.sv which four "squares" to set to the provided color and which four squares to set the background color, usually equivalent to the foreground squares of the previous frame. When a piece lands, the game logic clears the appropriate rows of its logical board and provides coordinates and signals for tetris.sv to clear the corresponding rows of its vram board. The description of the Game_Logic sv module goes into much further detail on the logic for the game itself.
The NIOS 2 is used to handle keyboard input, identical to the configuration from previous labs except here it uses on-chip memory instead of SDRAM. We do this because we instead use SDRAM to store our vram. The keys we use are W (rotate), A (move left), S (fall faster), and D (move right). We also use key 0 on the fpga to reset the game.
The timer.sv module keeps track of the game's timer which starts at 2 minutes and decreases down to zero, at which point the game ends. As that timer decreases the game also gets faster, which is handled by the game logic taking in the current time from time.sv as input. By also outputting this time to Color_Mapper.sv we're able to display the time on the screen. The score.sv module takes care of the game's score, taking in input from Game_Logic.sv and providing output to Color_Mapper.sv for it to display the score.

The memory that we chose to utilize for this project was the sdram present on the DE-10 Lite Shield using the sdram controller provided in the RTL SDRAM TEST by Intel that can be found online. Provided alongside this sdram controller was various megafunction instantiation such as for the fifo read buffer and the fifo write buffer, as well as the pll that was utilized by both sdram and its controller. The choice of sdram was done to demonstrate the utility of sdram which has far more storage than on chip memory present within the FPGA. The module tetris.sv manages all the data that needs to be written to/read from in sdram, by sitting between the sdram

controller and Game_Logic.sv, which has been described above. An internal state machine ensures that data is written to/read from sdram correctly, and that data does not get corrupted through timing violations. Most of the reading/writing to and from sdram is done in the vertical and horizontal blanking intervals to ensure that correct data is available when drawing to the screen.

SV Module Descriptions

Module: `Game_Logic.sv`
Inputs: `Reset, frame_clk, Clk, [7:0] keycode, [3:0] time_left [3]`
Outputs: `[6:0] blockXPos [4], [6:0] blockYPos [4], [6:0] blockXPrev [4], [6:0] blockYPrev [4], [15:0] blockColor, Clear_row, [3:0] Num_rows_to_clear, [6:0] Row_to_clear, [4:0] Score_to_add [4], [3:0] time_left [3]`
Description: The game logic is designed around the movement of a single “piece” (a collection of four “blocks” currently controlled by user input). A piece’s X and Y positions are updated in many places, mainly by the movement vectors `blockXMotion` and `blockYMotion` which are added to `blockXPosition` and `blockYPosition`. These additions are made when the corresponding frame counters `frame_count_move_X` and `frame_count_move_Y` reach the values specified by `frames_per_move_X` and `frames_per_move_Y`, the number of frames (cycles of `frame_clk`) in between X and Y movements respectively.
At every edge of the base 50MHz MAX10 clock, the user input keycode corresponding to the keycode of the key currently pressed on the keyboard is read. The four keys used by this logic are W (rotate piece), A (move left one square), S (increase vertical falling speed by reducing `frames_per_move_Y`), and D (move right one square). Keys A,D and W are only read when initially pressed, which is handled by setting a flag upon their press and unsetting that flag upon release and only acting when that flag goes from low to high.
Each four-block piece is stored as an unpacked array of four orientations stored as unpacked arrays of four pairs of (X,Y) coordinates for each block stored as 14-bit values with bits [13:7] holding the Y value and bits [6:0] holding the X value. These coordinates specify the layout of each orientation with the coordinates of each block relative to (0,0). Each piece has four orientations corresponding to four 90-degree rotations of the same piece. There are five pieces total.
The game board is maintained as an unpacked array of 20 10-bit registers, corresponding to the 10x20 grid of the tetris board. Each bit corresponds to a “square” of the board, with a 1 representing a square filled with a landed block and a 0 representing an empty square. The single moving piece has no impact on this board – this board only represents landed blocks. Only one piece is moving at a time. This position of each of the four blocks in the piece is maintained separately by the 4 pairs of registers `blockX`(1-4) and `blockY`(1-4).
Upon reset (driven by the `Reset` input), this code sets all locally used flags, coordinates, and counters to 0. It also resets the board to all zeros, and sets the current blocks’ coordinates to that of the first orientation of the first piece at the top center of the board. It sets `blockYMotion`

to its default value of 1, which is what keeps the blocks constantly falling. It finally sets the Y movement counter to its default speed `default_frames_per_move_Y1`.
Inside the always_ff block clocked by the 50MHz clock is a block that is driven by the 60Hz `frame_clk`. This is handled by conditioning on `frame_clk` being high and a flag (which we immediately set high in the block and reset when `frame_clk` goes low) being low, such that the block is only entered once on the positive edge of `frame_clk`.
In order for our module in charge of vram (tetris.sv) to properly erase and redraw the moving piece, Game_Logic.sv outputs the previous X and Y values of each block, `blockXPrevious`[0-3] and `blockYPrevious`[0-3]. In the frame clock block we first by default assign these previous coordinate values to the current coordinate values at the time of the positive edge (so before they are updated). Then we check if the `power_up` flag is set, in which case we increment the `color` index, creating a rainbow effect as the piece's color changes every frame. We check the game timer provided to us as input: if it is at 0:00, we set the `game_over` flag which disables most of the other logic. Otherwise, if it is less than 0:30 we change to the fastest speed for the Y frame clock. Otherwise, if it is less than 1:00 we change to the intermediate speed for the Y frame clock. This allows the game to get faster as the timer winds down.
When the moving piece collides, we save the Y coordinates of its four blocks as `blockYlanded`[0-3]. This is what we use to determine if a row should be cleared. In a conditional, we check if the forth block landed (`blockYlanded[3]` >= 1) and we check the status of that row of the game board. If that row is full of landed blocks (ten 1s), we set signals `clear_row`, `row_to_clear`, and `num_rows_to_clear` for tetris.sv to clear the vram, determining `num_rows_to_clear` based on number of rows above `blockYlanded`[3] that are also clear, up to a maximum of 4 including the initial row. We also set our clearing state machine to its initial clear state `clear1`, and similarly set the initial value of its rows to clear counter `clear_counter_start` based on the number of full rows above `blockYlanded`[3]. We start with `blockYLanded`[3] and repeat this whole clear check 3 more times for `blockYLanded`[2,1,0], each in a proceeding "else if" statement since our convention for the pieces to have the Y positions increase in order from block 0 to block 3.
Having entered the clearing state machine outside of the frame clock block, in `clear1` we first condition on the number of rows to clear to determine the `score_to_add` output for score.sv (1 point for 1 row, 5 points for 2 rows, 10 points for 3 rows, 25 points for 4 rows). If 2 or more rows are cleared we also initialize the power up by setting the `power_up` flag high and spawning the power up piece (piece 1, orientation 0). State `clear2` acts as an outer loop, and state `clear3` acts as the inner loop.The inner loop counter `clear_counter` starts at the row we currently want to clear and decreases down all the way to 0 as we copy each row of the game board to the row below it, effectively shifting the whole game board down by one row by the end. In state `clear2` we reset the inner loop counter `clear_counter` to its initial value `clear_counter_start` (the topmost filled row we want to clear) added to `clear_row_counter`, the outer loop counter which counts the number of rows we've cleared up to `num_rows_to_clear` which again ranges from 1 to 4. This way, of the up to 4 rows we want to clear we start with the topmost, clear that, shift the game board down, then move to the row below it if it's also a row we're clearing. Inside the inner loop state `clear3` we move each

row of the gameboard down, decreasing the counter until it is at 0 in which case we increase the outer loop counter `clear_row_counter` and move back to the outer loop state `clear2`. Along with the score and power up logic in `clear1`, we also initialize the inner loop counter to 0.

Back in the frame clock block, we also handle the rotate, move x, and move y blocks conditioned on the positive edge of counters `frame_count_rotate`, `frame_count_move_X`, and `frame_count_move_Y` respectively in the same fashion as the `frame_clk` condition. For the rotate block condition we also check that we're not in a move Y frame in which case we don't reset the counter, effectively queuing the rotate logic for the next frame. For the move Y block condition, the counter we use is compared to a register `frames_per_move_Y` whose value changes over time, effectively speeding up the Y movement as the game timer runs down.

In the rotate block we check if W has been pressed since the last rotation. If so, we check that a rotation would not produce any collisions. This collision condition uses 4 combinationally driven registers `blockX`(1-4)`Motion` and `blockY`(1-4)`Motion`. These are computed in an always_comb block to each be the difference between the current X/Y coordinates for a block and the X/Y coordinates of that block in the next orientation. When these differences are added to `blockX`(1-4) and `blockY`(1-4) respectively, the piece will effectively have rotated. This is exactly what we do if said rotation does not produce a collision. We also then update the variable representing the current piece orientation `block_orientation` as an index 0-3 into a piece's array of orientations.

In the move x block we reset the next X motion `blockXMotion` to 0. We then check if movement of the current piece by the `blockXMotion` vector would produce a collision. If not, we update the 4 blocks' positions `blockX`(1-4) by adding the same vector `blockXMotion` to the four. `blockXMotion` is driven by the pressing of the A and D keys, only used to move all blocks of a piece the same horizontal distance.

In the move y block we reset the next Y motion `blockYMotion` to 1. We then check for the case of a powerup.

If there's no powerup, we then check if movement of the current piece by the `blockYMotion` vector would produce a collision. If so, this means the piece has landed, and we then set the bits corresponding to the locations of the four blocks on the game board to 1. We set all of the previous Y values `blockYPrevious`[0-3] to 0 as a dummy value for the vram module to erase, since we don't want to erase the block that just landed. That is how our game works: vram stays persistent and only changes the screen where the single moving piece and clears where it moved from. Then once a piece lands the game logic tells the vram handler tetris.sv to not clear that piece from the screen, and it generates a new piece. Since a block has landed we also set `blockYLanded`[0-3] to the rows of each block that has landed. This is what the row clearing code uses to check for full rows once a block has landed. We also check if the row of any of the blocks is 0, meaning we've reached the top of the screen and set the `game_over` flag high, ending the game by deactivating most of the other logic. We generate a new piece by resetting `blockX`(1-4) and `blockY`(1-4) to another piece determined by indexing into the pieces array `Pieces`[`piece_count`], where `piece_count` is a counter that increments each 50MHz clock cycle providing pseudorandomness to the pieces. If this new piece is different from the current

piece, we go ahead and reset the coordinates. Otherwise, we pick the piece indexed after `piece_count`. In both cases we also set the `current_piece` register to the index of this new piece. We also increment the `color` palette index, making the next block a different color. If the Y movement doesn't produce a collision, we then update the 4 blocks' positions `blockY`(1-4) by adding the same vector `blockYMotion` to the four, exactly the same as with the X block.
That's what we do in the Y block if the `power_up` flag wasn't high. If it was high, then we'd similarly move the current piece and check for collisions, but instead of moving each block of the piece together and checking for any collisions, we move each block separately and check for their collisions on an individual basis. The power up works such that a horizontal long piece is generated such that each of the 4 blocks in the piece fall separately, such that if a block lands it doesn't stop the other blocks, effectively having the piece fill all the gaps below it. If an individual block collides we set its bit on the board to 1 and we set the corresponding `blockYPrevious` register to 0. We do not reset `blockX`(1-4) or `blockY`(1-4) unless all four blocks have landed. This way each of the four collision conditionals in this power up block are entered every time so that the `blockYPrevious` values are held at 0 (so that the vram handler doesn't actually clear them from the screen) and the blocks are not moved vertically. The blocks stay like this from the time they collide to the time the `power_up` flag goes low. That flag goes low when all four blocks have landed, which is determined by a conditional checking that all four blocks would produce collisions with their Y movements. Along with setting the flag low we also set all the registers that we would in the non-power up Y collision block including `blockYPrevious`[0-3], `blockYLanded`[0-3], `blockX`(1-4) and `blockY`(1-4) reset to a new piece, `current_piece`, and `block_orientation`.
Lastly, inside of an always_comb block we assign our local registers to their respective output signals.
Purpose: Holds the entirety of the logic for the game itself.

Module: `font_rom.sv`
Inputs: `addr[10:0]`
Outputs: `data[7:0]`
Description: holds the data for the character glyphs used in this lab. Data is read only and is read byte by byte, or row by row of each character.
Purpose: Holds the glyphs used to draw characters to the screen via VGA.

Module: `timer.sv`
Inputs: `clk, reset`
Outputs: `[3:0] time_left [3]`
Description: This module takes the frame clock as opposed to the MAX10 50MHz clock. Upon `reset`, this module sets the timer to 2 minutes and immediately starts counting down. `[3:0] time_left [3]` are unpacked registers that represent the 1-digit minute and 2-digit seconds that are to be displayed in game. The value the timer outputs changes once every 60 frame clocks.
Purpose: 2 minute timer to keep track of how much time is left in the game.

Module: score.sv
Inputs: clk, reset, [4:0] score_to_add [4]
Outputs: [4:0] digits [4]
Description: This module accepts the typical system clock and reset, where upon reset the score is cleared. [4:0] score_to_add [4] is an array that holds the digits of the score to add to the current score on screen. These digits are expected to be in the range of 0-9, and the new score is updated based on this assumption. The output [4:0] digits [4] is the up-to-date score with a similar format to its input. The new score is calculated combinationally, and the result is stored on the next positive edge, and so [4:0] score_to_add [4] must only be provided for one cycle of the base clock.
Purpose: Holds the score for the game, keeps track of points accumulated.

Module: Color_Mapper.sv
Inputs: Clk, hs, reset, frame_clk, [6:0] clear_row, [4:0] score_to_add [4], [4:0] digits [4], [3:0] gameClock [3], [15:0] Row [10], rowRead, [9:0] DrawX, [9:0] DrawY
Outputs: [7:0] rowNum, LD_Row, [7:0] Red, [7:0] Green, [7:0] Blue
Description: Color_Mapper draws the current state of the game to screen for each frame. While this module is drawing in a given frame, it will load the next row that it is about to draw when it is in the horizontal sync interval and the next line corresponds to the next row on the board (the game has 20 rows). It then takes the data returned from sdram and draws it to the board based on the current [9:0] Draw X and [9:0] DrawY. When Color Mapper (by this I mean [9:0] DrawX and [9:0] DrawY) is somewhere in the bounds of the screen, it will first check to see if we are in the part of the screen that pertains to the game board and if so, we draw the color as obtained from [15:0] Row [10], with black boxes around the blocks, indexing this array of unpacked registers by the current column of the gameboard that we are at. If we are in the region of the screen dedicated to the scoreboard (top right of the gameboard), we draw the score based on [4:0] digits [10], using font_rom.sv that was provided in an earlier lab, using similar indexing logic. When reading from font_rom.sv we draw the foreground when encountering a 1 and the background when we encounter a 0. When we are in the region reserved for the game timer (top left of the gameboard), we again use font_rom.sv, except that this time we draw the values stored in [3:0] gameClock [3]. The last region that we check for in color mapper is if [9:0] DrawX or [9:0] DrawY is where the pointer adder is supposed to be displayed based on the row that is cleared. For example, if only the second from the bottom row is cleared, then a +01 will show up just to the right of the row outside of the bounds of the game. Finally, if none of the above conditions are met, then we draw a generic background color, which for this project was chosen to be green.
Purpose: Handles logic for drawing the game to the screen via VGA.

Module: tetris.sv
Inputs: clk, vs, hs, key, reset, row_ld, [7:0] clear_row, [7:0] clear_num_rows, clear_the_row_ho, [7:0] row, [6:0] preX [4], [6:0]

preY [4], [6:0] postX [4], [6:0] postY [4], [9:0] DrawX, [9:0] DrawY, [15:0] wr_buffer, [15:0] rd_buffer, [15:0] readdata, [15:0] color
Outputs: row_ready, write_req, read_req, write_ld, read_ld, [24:0] writeaddr, [24:0] readaddr, [15:0] writedata, [7:0] Red, [7:0] Green, [7:0] Blue, [3:0] hex_num_4, [3:0] hex_num_3, [3:0] hex_num_1, [3:0] hex_num_0, [15:0] read_reg[10]
Description: The first 200 addresses in sdram are used to hold the color of the blocks for the 10x20 board, where each address just holds once color. The addresses are indexed in row-major order when accessing for the reads and writes. The information encoded at each of the address locations takes the form of x0ABC, with each letter representing a 4-bit red, green, or blue in that order. Upon reset, the internal state machine that handles the reads/writes goes through a 5 state initialization phase whereby the background color (blue) is written to each of the 200 addresses. In Init_RAM1, we set write_ld high and update writeaddr so that it takes the current value of an initialization counter as the address to write to. In Init_RAM2, write_ld goes low, and at this point the address we want to write to has been loaded by the sdram controller. write_req is then set high and writedata is loaded with the background color. In Init_RAM3, write_req is set low, and at this point, the data that we wish to write has been captured by the sdram controller. The sdram controller is configured for a write length of one, and so each address location must be written to one at a time. The state machine then waits two clock cycles to allow the write fifo buffer to be filled with the data we wish to write. In Init_RAM4, we wait until the write fifo buffer empties before moving on. In Init_RAM5, we check to see if we have written to all 200 address locations by checking the initialization counter. If we have, we move to Hold1, otherwise we increment the initialization counter and then return to Init_RAM1.

Hold1 is the primary wait state that the state machine resides in while waiting to perform other operations after the sdram has been initialized. In this state, we first if vs and clear_the_row_ho are both high and that clear_already is low. Essentially, if we are in the vertical sync interval and we have a signal to clear a particular row and such hasn't happened already in the current interval, we set clear_already high and capture clear_row to be used as an index for clearing the screen. We then move through the states designated to clear the screen starting at a specific row, where the first state is MemRead1. The next condition we check for in this state is if vs is high and update_flag is low, then we set update_flag high along with clear_flag before moving through the states that handle clearing the old blocks locations in sdram and writing the new ones, starting at PWA. The next condition that is checked is if row_ld is set high, and if it is, we readout a row and then output it to color_mapper for drawing. Finally, we check if vs, and if it is, then we set update_flag and clear_already low so that we can invoke screen clearing or block writing in the next vertical sync interval.

In the event that we branch to MemRead1, we first wait several clock cycles, then we reset row_counter and write_counter to zeroes, where the former keeps track of the column we are writing to, and the latter is used to buffer states so as to not violate sdram controller

timings. Additionally, read_ld is set high and the readaddr is set to the address of the first block in the row we are clearing minus the number of rows we are instructed to clear. In the next state ReadMem2, read_ld is set low, and it is at this point that the address of the first element in the row is locked into the sdram controller. We then wait for the read fifo buffer to fill up to 10 elements (the length of one row), before setting read_req high. The sdram controller is configured to have a read length of 10, and so we must allow the buffer to read out 10 elements at a time before emptying it. In MemRead3, an internal [15:0] row_reg [10] captures the first element in the row and stores it in a register. In MemRead4, we continue to readout the buffer until it is empty, storing the data in consecutive unpacked registers in row_reg. After the buffer is emptied, we move to MemWrite1. In MemWrite1, we set write_ld high and writeaddr to the address of the row we are clearing offset by the row counter to access the correct address since we are writing each location once at a time. In MemWrite2, we set write_ld to low to have the address loaded, and then write_req is set high and writedata is loaded with the element of row_reg that has the same column index as location in memory we are writing to. In MemWrite3, write_req is set low so that writedata is captured. After a two cycle buffer, we transition to MemWrite4 where we wait for the write fifo buffer to empty completely. Finally, in MemWrite5, We check to see if we have cleared all the rows we needed to clear, and if we have, we transition to Hold1. Otherwise, we check to see if we have reached the end of the row that we are writing to, and if we have, we reset row_counter and get the next row before moving to MemRead1. Otherwise, we simply increment row_counter and go to MemWrite1.

In the event that we branch to PWA, we begin the process of clearing the four previous blocks by writing the background color to their location in vram, and then write the block color at the new locations. In PWA, we set write_ld high and check to see if we a clearing by checking that clearing_flag is high. If it is, then writeaddr gets the computed address of the first of the four blocks pre_block_addr[0], otherwise writeaddr gets post_block_addr[0]. In the next state WA, write_ld is set low to capture the address we are writing to, and write_req is set high. writedata is then loaded with the background if clearing_flag, otherwise we load the block color. In the next state QWA, we set write_req low to capture the data, and then wait two clock cycles before moving to FWA. In FWA, we wait for the write fifo buffer to empty before moving to PWB. PWB onwards follows a similar execution to that of PWA to FWA. This execution cycle is repeated for the next 3 blocks of the pieces. Once we reach the final state, we move to Hold2 where we set clearing_flag low, and proceed through another four random writes in order to write the new positions of the blocks. After this has been completed, we return to Hold1.

In the event that we transition to PRA in order to read an entire row to output, we first buffer 8 clock cycles to avoid violating sdram timings before setting read_ld high and loading readaddr with the address that was requested externally. It is also at this point that write_counter is reset before moving to the next state. In RA, read_ld is set low to capture the address to start loading from. We then wait until the read fifo buffer is filled up with

10 elements before `read_req` is set high to start empting the buffer. In the next state `RAI`, we capture the first element returned in `readdata` and store it in the first unpacked register of `readdata_reg`. In the next state `FRA`, we check to see if the read buffer has been emptied, and if not, then we keep reading `readdata` into `readdata_reg` indexed by an incrementing `write_counter`. Note that `readdata` pumps out the next element in the read fifo buffer on each positive edge. Once the buffer has been emptied, we set `row_ready` high to indicate to the drawing logic that the current row can be read from. We additionally reset `write_counter` and `read_req` to low before returning to `Hold1`.

Outside of the state machine, some combination logic does address computation on the provided block in order to correctly index vram when clearing/writing a block that is falling.

Purpose: Maintains the state machine that allows us to read from/write to sdram, which is utilized as vram for drawing to the screen.

Module: `command.sv`
Inputs: `CLK, RESET_N, [`ASIZE-1:0] SADDR, NOP, READA, WRITEA, REFRESH, PRECHARGE, LOAD_MODE, REF_REQ, INIT_REQ, PM_STOP, PM_DONE`
Outputs: `REF_ACK, CM_ACK, OE, [`SASIZE-1:0] SA, [1:0] BA, [1:0] CS_N, CKE, RAS_N, CAS_N, WE_N`
Description: Included in the sdram controller provided by DE-10 LITE SDRAM RTL TEST. Interacts directly with the sdram datapath module as well as the sdram itself, with an internal state machine that monitors the state of the sdram When the sdram is not busy, it will issue the appropriate command..
Purpose: Takes various SDRAM operations (nop, precharge, etc) and sends out the appropriate control signals based on the provided operations to execute.

Module: `sdr_data_path.sv`
Inputs: `CLK, RESET_N, [`DSIZE-1:0] DATAIN, [`DSIZE/8-1:0] DM`
Outputs: `[`DSIZE-1:0] DQOUT, [`DSIZE/8-1:0] DQM`
Description: Included in the sdram controller provided by DE-10 LITE SDRAM RTL TEST. Aligns the input and output data for the SDRAM control path.
Purpose: Aligns DQM correctly for the SDRAM datapath.

Module: `control_interface.sv`
Inputs: `CLK, RESET_N, [2:0] CMD, [`ASIZE-1:0] ADDR, REF_ACK, INIT_ACK, CM_ACK`
Outputs: `NOP, READA, WRITEA, REFRESH, PRECHARGE, LOAD_MODE, [`ASIZE-1:0]SADDR, REF_REQ, INIT_REQ, CMD_ACK`
Description: Included in the sdram controller provided by DE-10 LITE SDRAM RTL TEST. Generates the appropriate commands based on requests sent into this module.
Purpose: Generates SDRAM operations for the command module.

Module: `Sdram_Control.sv`

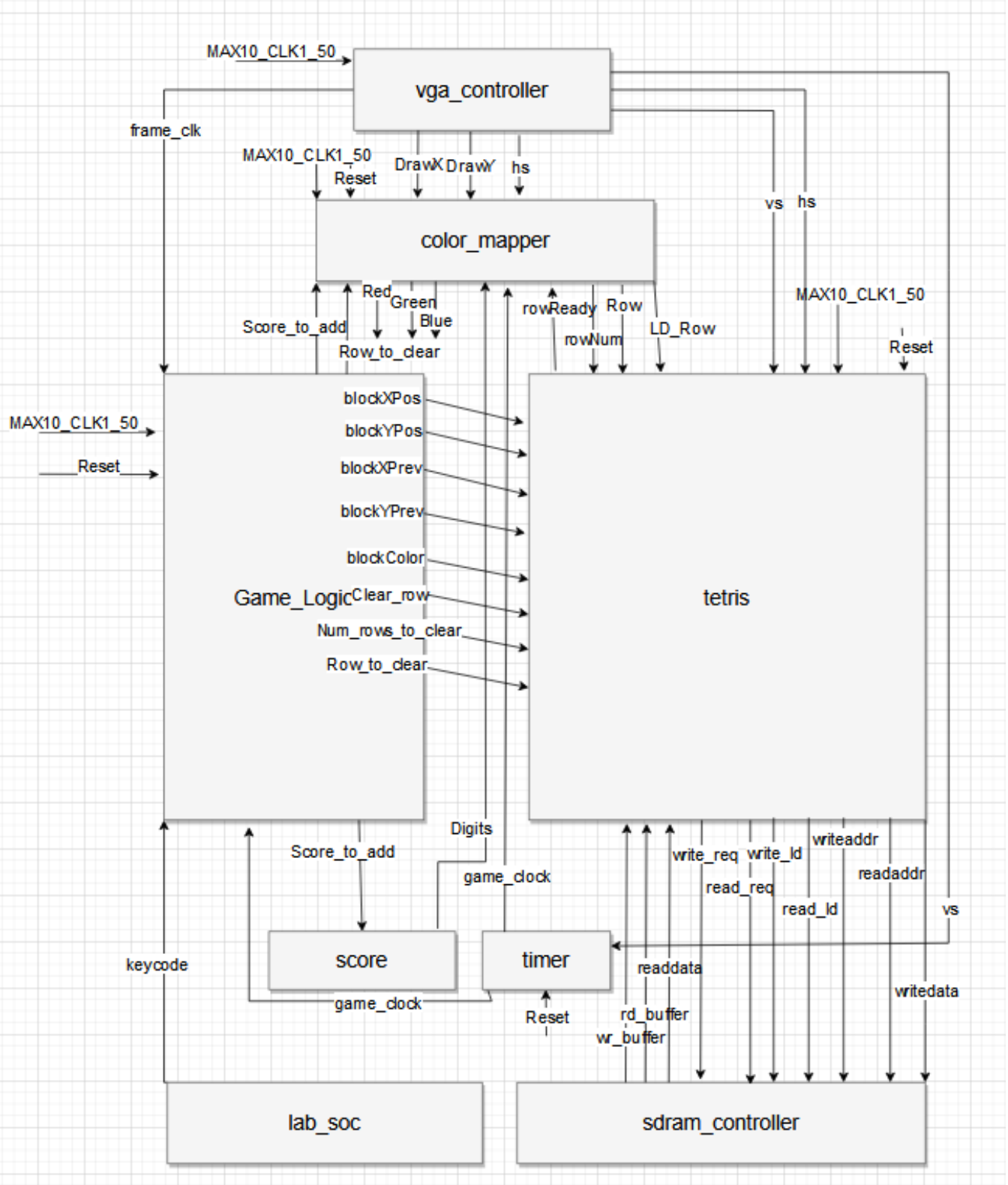
Inputs: `REF_CLK, RESET_N,[`DSIZE-1:0] WR_DATA, WR, [`ASIZE-1:0] WR_ADDR, [`ASIZE-1:0] WR_MAX_ADDR, [8:0] WR_LENGTH, WR_LOAD, WR_CLK, RD, [`ASIZE-1:0] RD_ADDR, [`ASIZE-1:0] RD_MAX_ADDR, [8:0] RD_LENGTH, RD_LOAD, RD_CLK, [`DSIZE-1:0] DQ`

Outputs: `WR_FULL, [15:0] WR_USE, [`DSIZE-1:0] RD_DATA, RD_EMPTY, [15:0] RD_USE, [`SASIZE-1:0] SA, [1:0] BA, [1:0] CS_N, CKE, RAS_N, CAS_N, WE_N, [`DSIZE-1:0] DQ, [`DSIZE/8-1:0] DQM, SDR_CLK`

Description: Included in the sdram controller provided by DE-10 LITE SDRAM RTL TEST. This module maintains the fifo read buffer and fifo write buffer that is attached to the read side of the controller and the write side of the controller respectively.

Purpose: Top level module for the sdram controller that directly communicates to the sdram pinout in the top level file of the project.

Block Diagram



Design Resources and Statistics

The following tables contain the design resources and statistics calculated for the final implementation of the game:

LUT	14014
DSP	0
Memory (BRAM)	1363968
Flip-Flop	6894
Frequency	59.02MHz
Static Power	103.22mW
Dynamic Power	144.70mW
Total Power	282.99mW

Table 1: Statistics of Resources Used

Notably, a large portion of the BRAM available on the FPGA is utilized, which is in large part because NIOS was switched to use on-chip memory so that the sdram controller could be used by our project.

Conclusion

Overall our design worked as planned. There were a few hurdles that we had to overcome. One such hurdle was the issue of changing registers in different clock domains in Game_Logic.sv. At first we had four different always_ff blocks using five different clocks: MAX10_CLK1_50, vs as the frame clock, and the move x move y and rotate clocks which we generated based on the frame clock. But then we ran into the issue of certain registers needing to be written to in these different always_ff blocks, which isn't allowed since a register can only be clocked at one frequency. We then realized we could restructure our code into a single always_ff block clocked at the fastest of the five, MAX10_CLK1_50. Then we would use counters and conditionals within that main block to have our separate moving and rotating logic occur at different intervals while all writing to the same 50MHz clocked registers. This allowed us to write to the same registers under different pseudo clock domains.

Another portion of the project that was difficult to implement was managing the inputs to the sdram controller so as to not violate timings. For example, if we attempted to read out from the read fifo buffer before the data has a chance to get properly locked, the readout could be corrupted by as little as 1-bit. A similar effect would happen to improper writes, and so to mitigate this buffer wait states had to be implemented to allow time for the controller to "catch up", which was done through analyzing sdram timings in Signal Tap.