ECE 385

Spring 2024

Final Project

SystemVerilog Implementation of Tetris

Connor Tan & Krish Sahni

TA: Henry Gillespie

Introduction:

For our final project, we decided to design and implement the classic nintendo game, Tetris on our FPGA, with keyboard functionality to interact with it. Along with the Spartan-7 FPGA, we will use the MicroBlaze processor, the MAX3421E, and the VGA to HDMI module. The MAX3421E is used so that we can use a keyboard peripheral to control the game. The VGA to HDMI module is used so that we have an output on the screen. We will base our MicroBlaze peripheral setup on what we did on lab 6.2.

In our game of Tetris, we plan on implementing moving colored blocks, a bounds checker, and a score counter based on the number of rows cleared, like the original game of Tetris. The user will be able to press the a and d keys to move the blocks left and right, the w key will be used to rotate the block clockwise, and the s key will be used to have the block go down the screen faster. We also implemented a start screen that shows once your program the device, and an end screen that shows once you lose the game. Spacebar is used to start the game and reset the board.

Description of MicroBlaze System:

The MicroBlaze system that we used in week 2 consisted of the MicroBlaze processor, which was set to the microcontroller preset, as well as added peripherals that allowed us to interact with the MicroBlaze. This added modules that are required for the MicroBlaze processor to work as a microcontroller. These modules are the MicroBlaze local memory, the processor system reset module, the clocking wizard, the debug module, the AXI timer module, the AXI Interconnect module, the concatenate module, and the AXI interrupt module. We also added the AXI Uartlite peripheral, 3 AXI GPIO peripheral, and the AXI Quad SPI peripheral.. Figure 2 shows the block diagram for the system for lab 6.2 that we used for our Tetris inputs and screen output.

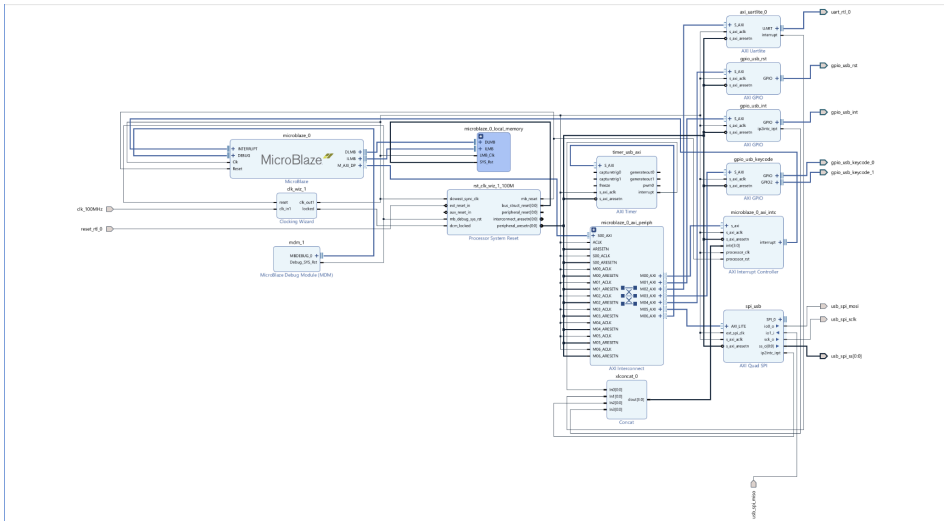

Figure 1 - Block Diagram of the MicroBlaze System of Lab 6.2

MicroBlaze Processor:

The first, and main component of the system is the MicroBlaze processor itself (figure 2). As mentioned before, this component is a soft-IP 32-bit RISC, modified harvard processor that is highly configurable using a high level programming language.

Figure 2 - MicroBlaze Processor in Block Diagram

Clocking Wizard:

This block takes in the clock signal output and inputs that signal into the rest of the connected components. This is to ensure that all the clocks are synchronized. It takes in a 100 MHz signal and outputs two signals, a 25 MHz signal and a 125 MHz signal. You can see how it is configured in our block diagram in figure 3.

Figure 3 - Clocking Wizard in Block Diagram

AXI Interconnect:

This block connects multiple AXI masters to one or more AXI Slaves. We need this block because we have multiple slaves that we need to connect to. This block can be seen in figure 4.

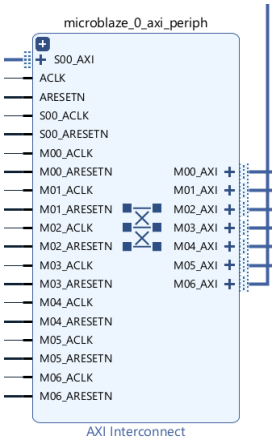

Figure 4 - AXI Interconnect in Block Diagram

AXI Quad SPI:

This module connects the AXI Interface to the slaves that support the SPI protocol. In our case, it's the USB. It bridges the AXI interface with SPI-compatible devices, enabling data exchange with the USB controller. It allows precise control and configuration to ensure optimal communication efficiency within our SoC architecture. Figure 5 shows the block diagram of this module.

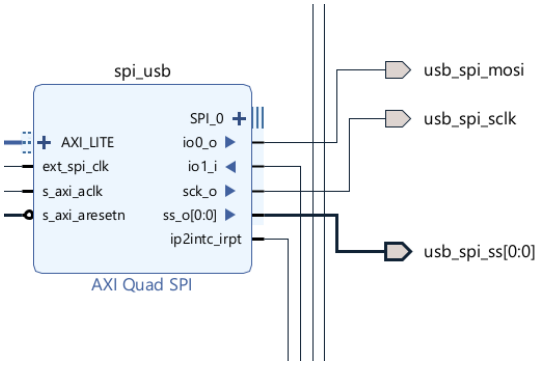

Figure 5 - AXI Quad SPI in Block Diagram

AXI Uartlite:

This module connects to the processor and is used for asynchronous serial communication and data transfer. It facilitates synchronous serial communication between the processor and external devices, supporting essential data transfer functions. It is mainly used to streamline debugging and peripheral interactions. Figure 6 shows the block diagram of this module.

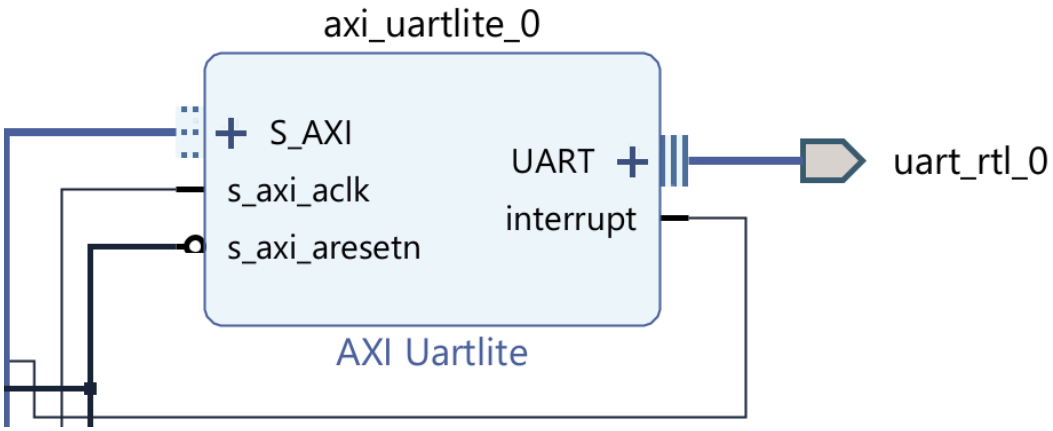

Figure 6 - AXI Uartlite in Block Diagram

AXI GPIO:

These blocks are used to connect the peripherals used on the Urbana board, like the LEDS, the USB, and the buttons, with the rest of the system. Figure 7 shows the block diagrams of these modules for this lab.

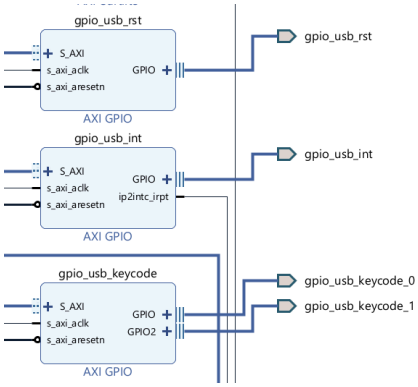

Figure 7 - AXI GPIO modules in Block Diagram

AXI Interrupt Controller:

This controller takes on the multiple interrupt signals in the system and concentrates them into one interrupt signal that gets put back into the main MicroBlaze Processor. Figure 8 shows the AXI Interrupt Controller in the block diagram.

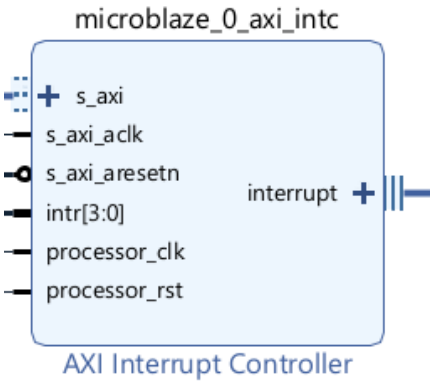

Figure 8 - AXI Interrupt Controller Module in Block Diagram

Processor System Reset:

This module allows the user to set the parameter to enable or disable the system. It initializes and resets the processor and peripheral devices and clears the memory of the system. Figure 9 shows the Processor System Reset block in the diagram.

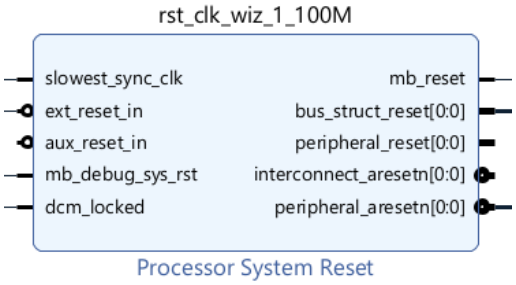

Figure 9 - Processor System Reset Module in Block Diagram

AXI Timer:

This module acts as the timer interface to the AXI-4 Lite and provides the signal to AXI Reset and AXI clock. Figure 10 shows the AXI Timer diagram.

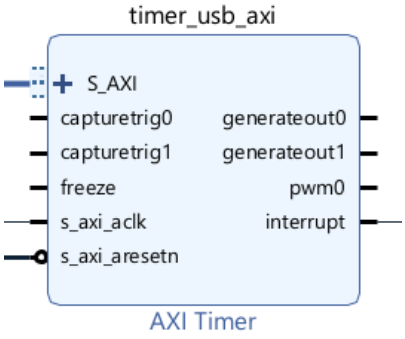

Figure 10 - AXI Timer Module in Block Diagram

Concat:

This module concatenates all the interrupt signals in the system and feeds it into the interrupt controller. Figure 11 shows this module in the block diagram.

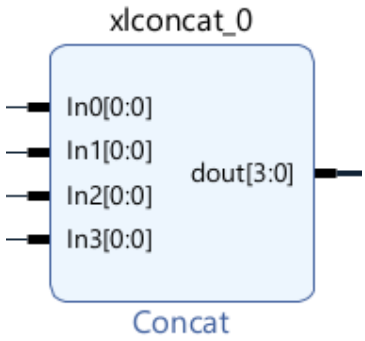

Figure 11 - Concatenate Module in Block Diagram

MicroBlaze Local Memory:

This block holds all the local memory within the MicroBlaze system. Figure 12 shows this module in the block diagram.

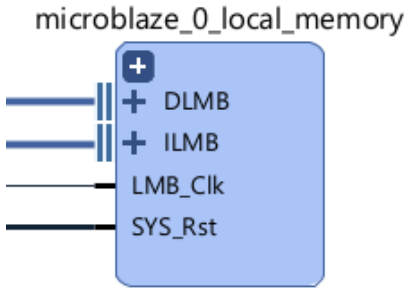

Figure 12 - MicroBlaze Local Memory in Block Diagram

MicroBlaze Debug Module:

This module enables the debugging for the MicroBlaze system. This module uses JTAG-based debugging. Figure 13 shows the module in the block diagram.

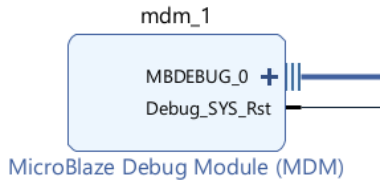

Figure 13 - MicroBlaze Debug Module (MDM) in Block Diagram

Description of .SV Modules:

Module: *mb_usb_hdmi_top.sv*

Inputs: Clk, reset_rtl_0, [0:0] gpio_usb_int_tri_i, usb_spi_miso, uart_rtl_0_rxd
Outputs: 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, hdmi_tmds_data_p), [7:0] (hex_segA, hex_segB), [3:0] (hex_gridA, hex_gridB)
Description: This is the top level module for the mb_usb_hdmi block design.
Purpose: This is the module that is called during synthesis that will be used as the hardware to program the device.

Module: *hex_driver.sv*

Inputs: clk, reset, [3:0] in[4]
Outputs: [7:0] hex_seg, [3:0] hex_grid
Description: The hexdriver for the LED outputs on the physical Spartan-7 FPGA board.
Purpose: This module is used so that we can see what we are actually doing on the FPGA board. It is needed to display the value loaded into Register B and the final product.

Module: vga_controller.sv

Inputs: pixel_clk, reset
Outputs: hs, vs, active_nblank, sync, [9:0] (drawX, drawY)
Description: The VGA controller for a 640x480 display.
Purpose: This module is designed to generate synchronization signals and pixel coordinates for a 640 x 480 VGA display.

Module: ball.sv

Inputs: Reset, frame_clk, [7:0] keycode, [3:0] current_x_pos, [4:0] current_y_pos, [219:0]fallen_pieces, [1:0] cur_rot
Outputs: [9:0] (BallX, BallY, BallS), [3:0] new_x_pos, [4:0] new_y_pos, [219:0] new_fallen_pieces, collision_detected, intersection_bottom, [7:0] (test_blk_1, test_blk_2, test_blk_3, test_blk_4), [2:0] (test_width, test_height), [1:0] new_rot, remove_row_en, [4:0] remove_row_y, game_over_en, [2:0] cur_piece
Description: Handles user inputs on moving blocks, monitors game states, and updates game variables such as positions, rotations, and collision states.
Purpose: Robust control mechanism to handle the block rotations, collisions, and row clearing, based on real time inputs from the user.

Module: calc_test_pos_rot.sv

Inputs: frame_clk, [7:0] keycode, [2:0] timer, [3:0] cur_pos_x, [4:0] cur_pos_y, [219:0] fallen_pieces, [7:0] last_keycode, key_action_taken, [2:0] cur_piece, [1:0] cur_rot, [2:0] (width_check, height_check), drop_mode
Outputs: [3:0] test_pos_x, [4:0] test_pos_y, [1:0] test_rot, test_intersects, [2:0] (test_width, test_height), test_drop_mode

Description: This module simulates potential positions and orientations of the game pieces based on player inputs and internal game logic, adjusting for boundary conditions and collision scenarios.

Purpose: This module aims to pre-calculate and validate the movement and rotation of game pieces to ensure they do not overlap with existing elements on the game board, facilitating smooth and error-free gameplay interactions.

Module: calculate_cur_block_pos.sv

Inputs: [2:0] piece, [3:0] pos_x, [4:0] pos_y, [1:0] rot

Outputs: [7:0] (blk_1, blk_2, blk_3, blk_4), [2:0] (width, height)

Description: The calculate_cur_block_pos module computes the positions of blocks based on the current piece type, position, and rotation, adjusting for various orientations and ensuring the blocks fit within the predefined game grid.

Purpose: This module is designed to dynamically determine and update the positions of tetris-like game pieces during gameplay, facilitating collision detection and rendering on a grid-based game board.

Module: next_piece_generator.sv

Inputs: clk, reset

Outputs: [2:0] next_piece

Description: The next_piece_generator module utilizes a linear feedback shift register (LFSR) to generate a pseudo-random sequence, determining the next piece type for a Tetris-like game based on the current state of the LFSR.

Purpose: This module is designed to continuously provide new, randomly selected Tetris piece types, ensuring variety and unpredictability in the game sequence, thereby enhancing gameplay dynamics.

Module: complete_row.sv

Inputs: clk, [219:0] fallen_pieces

Outputs:[4:0] row, enabled

Description: The complete_row module checks for full rows within the game grid represented by a 220-bit array, identifying rows that are completely filled with blocks.

Purpose: This module is designed to signal when a row in the game is fully occupied, enabling game logic to clear the row and possibly trigger scoring, thus supporting gameplay mechanics that involve row completion and clearing.

Module: color_mapper.sv

Inputs: remove_row_en, [9:0] (BallX, BallY, DrawX, DrawY, Ball_size), [219:0] fallen_pieces, clk, [7:0] (cur_blk_1, cur_blk_2, cur_blk_3, cur_blk_4,) [1:0] outputState, [2:0] cur_piece

Outputs: [3:0] Red, Green, Blue

Description: The color_mapper module processes various inputs to determine the color outputs for a graphical interface in a game, manipulating RGB values based on game

logic conditions and the current state of the game, such as active pieces, game board edges, and completed rows.

Purpose: This module is designed to dynamically render the start and end screen, and the game board and its elements, adjusting colors for active game pieces, completed lines, and other graphical elements.

Module: fsm.sv

Inputs: [7:0] keycode, gameOver, Clk, Reset

Outputs: [1:0] outputState

Description: The fsm (Finite State Machine) module defines state transitions for a game based on user inputs and game conditions, cycling through initial, active, and termination phases.

Purpose: This module governs the operational flow of a game, transitioning between game states such as starting the game, ongoing gameplay, and ending the game, based on player interactions and game outcomes, enhancing the structure and user experience of the game.

Block Diagrams for SystemVerilog Implementation of Tetris

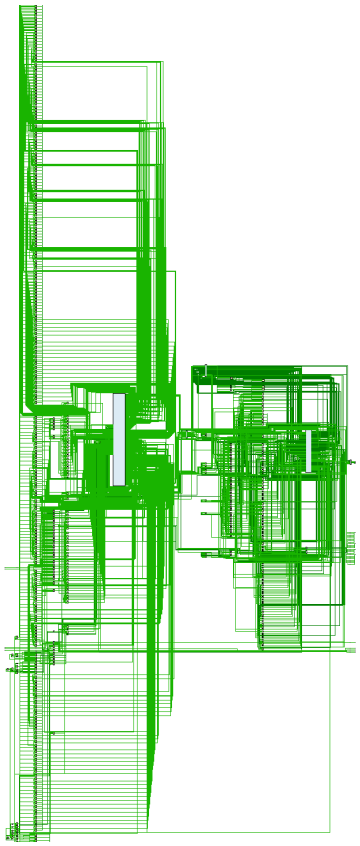

Figure 14 - Whole Top Level Diagram of Tetris Implementation

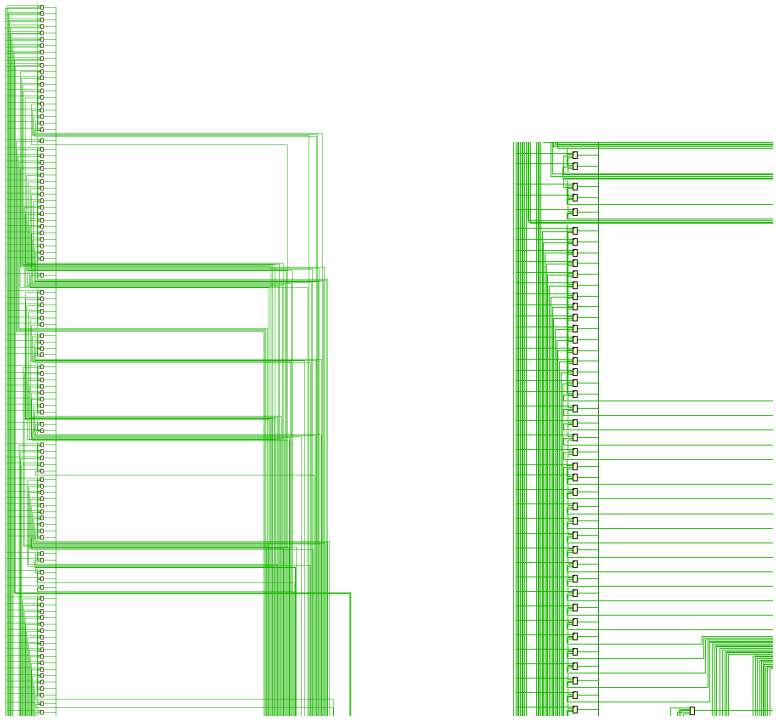

Figure 15 & 16- fallen_pieces registers

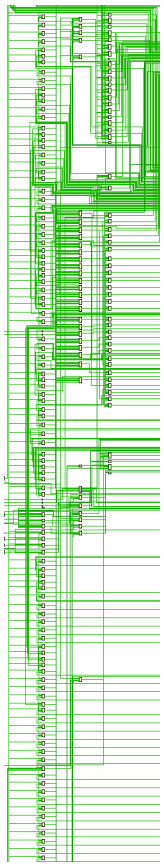

Figure 17 - shifting_row LUT

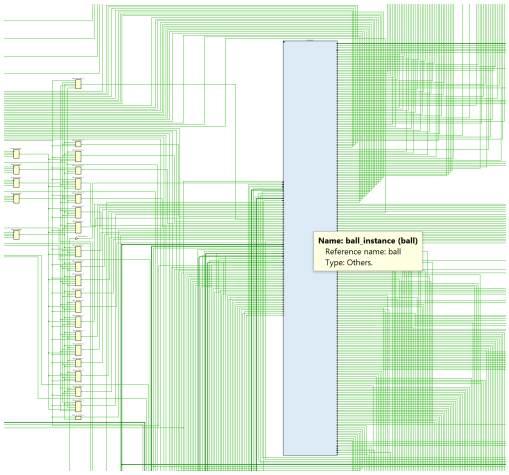

Figure 18 = Ball instance module

Figure 19 - VGA controller module with color registers on the left

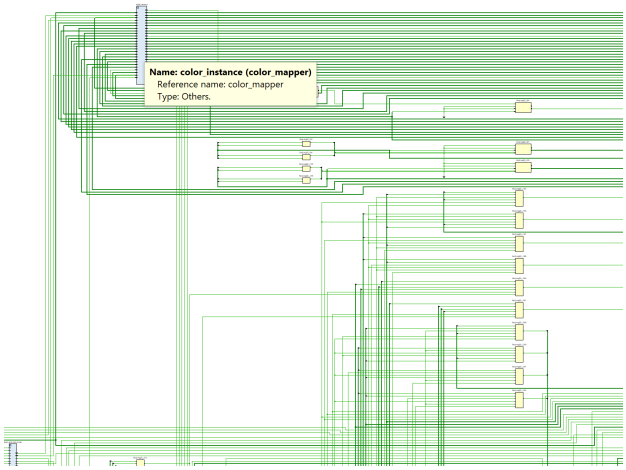

Figure 20 - color_mapper module with color registers on the right

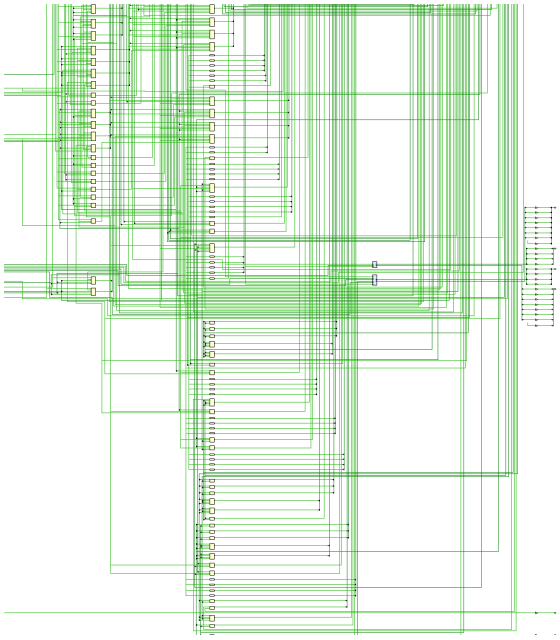

Figure 21 - Hex driver modules on the right, color registers on the left.

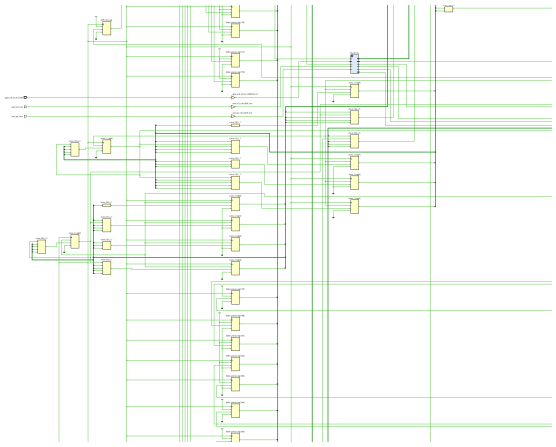

Figure 22 - Score registers (left) and mb_block module (right)

Description of SystemVerilog Implementation of Tetris.

Collision Detection

The collision detection in this Tetris implementation is managed by the `calc_test_pos_rot` module, which plays a pivotal role in the game's dynamic environment. It pre-calculates potential positions and rotations for the falling Tetris pieces in response to player inputs. This proactive simulation allows the module to predict and validate each move before it occurs, ensuring that placing a piece does not result in an overlap with existing blocks or extend beyond the game's designated boundaries. This preemptive checking mechanism helps maintain the integrity of the game by preventing illegal moves, thereby ensuring that all placements are validated before they are committed to the game board. This module is crucial for supporting the game's fluidity and responsiveness, providing a seamless interaction where the outcome of each move is instantly known and adjusted for.

Rotations

The `calculate_cur_block_pos` module underpins the rotation functionality in Tetris, providing players the ability to rotate pieces to better fit them into the playing field. It adjusts the blocks' positions based on the current rotation state of a piece, updating each block's position in real-time as the piece rotates. This ensures that the orientation of pieces is dynamically recalculated to accommodate new orientations within the confines of the game board and in relation to other pieces. The module ensures that each rotation is physically possible, avoiding overlaps and out-of-bound errors, which is essential for maintaining the flow and strategic depth of the game. By enabling precise and responsive rotations, this module greatly enhances player control and strategic options, making the gameplay experience more engaging and versatile.

Game Start and End

The `fsm` (Finite State Machine) module orchestrates the overall flow of the game, from initiation to conclusion. It manages the different states of the game—starting, playing, and ending—seamlessly transitioning between them based on player actions and game conditions. Initially, the game waits for a player input, such as pressing the spacebar, to transition from the start screen to the gameplay state. This module also monitors for the game over condition, which occurs when the Tetris pieces stack up to the top of the playing field, and then transitions to the end game state, which displays a score recap or end screen. By modularizing these states, the `fsm` facilitates easy management of the game's phases, enhancing user interaction and ensuring a smooth transition that maintains player engagement from start to finish.

Scoreboard

Although not directly implemented in the provided modules, a scoreboard feature would be essential for tracking player progress and enhancing competitive gameplay. Typically, it would monitor and update scores based on the number of lines cleared at a time, factoring in the level

of difficulty which increases as the game progresses. The `complete_row` module, which detects and handles row completions, would trigger score updates, rewarding players for clearing multiple lines simultaneously with higher scores. This would incentivize players to play strategically, enhancing the depth and replayability of the game.

Row Checking

The `complete_row` module is specifically engineered to monitor for complete rows across the Tetris grid. It methodically checks each row to determine if it is fully occupied by blocks, using a logical AND operation to assess the occupancy of all blocks in a row. If a row is completely filled, the module triggers a process to clear the row, thereby maintaining gameplay flow and opening up space for new pieces. This automated checking is critical for sustaining game progression and provides a consistent mechanism for updating the game state in response to player actions.

Row Clearing

Following the detection of a complete row, the game must then clear this row and adjust the placement of all blocks above it. This process, potentially extending the functionality of the `complete_row` module or through a dedicated system, involves shifting all rows above the cleared row downward to fill the vacant space. This not only updates the game's state but also increases the challenge and pace as players progress, forcing them to adapt to the dynamically changing game board. This row clearing process is vital for continuing the game, allowing for the accumulation of points and prolonging the gameplay experience by preventing an early game over.

Immediate Block Drop Logic

Our Tetris implementation features an immediate block drop, controlled by the calc_test_pos_rot module, which enhances gameplay by allowing players to quickly drop a piece to the lowest available position within its column using the 's' key. This rapid drop function calculates the furthest position the piece can reach without collision by iteratively testing each downward spot until an overlap occurs. Once detected, the piece locks into the last clear position, allowing for swift, strategic placements and accelerating the pace of the game. This feature is particularly beneficial for experienced players who need to make quick decisions, optimizing their gameplay experience by minimizing delays.

Design Resources and Statistics

LUT	5142
DSP	3
Memory (BRAM)	8
Flip-Flop	2974
Latches	80
Frequency	0.5571 MHz
Static Power	.075 W
Dynamic Power	.388 W
Total Power	0.463 W

Conclusion:

In conclusion, our final project for ECE 385 demonstrated the implementation of the classic Nintendo game, Tetris, on an FPGA platform. Working together, we leveraged the Spartan-7 FPGA integrated with a MicroBlaze processor to bring this project to fruition.

Throughout this project, we explored and applied various aspects of digital design, from basic logic gates to complex system integrations involving the MicroBlaze processor, interfacing with peripherals through AXI protocols, and handling VGA output through custom SystemVerilog modules. Our approach allowed us to deepen our understanding of embedded system design, particularly in how software and hardware can be seamlessly integrated to create interactive and responsive systems.

We implemented key game mechanics such as piece rotation, collision detection, row clearing, and a scoring system—all orchestrated through our custom modules. These elements were critical in replicating the authentic Tetris gameplay experience. We managed to incorporate effective controls using a keyboard interface, allowing smooth and intuitive gameplay which was a significant achievement given the constraints of hardware programming.

Moreover, our design included a start screen and an end screen, enhancing the user interface and making the game more user-friendly. The successful implementation of these features was not only a technical achievement but also a creative one, showcasing our ability to design a complete and functional game system.

The challenges we faced, particularly in optimizing the logic for speed and efficiency, and debugging the system in real-time, provided us with invaluable learning experiences. These challenges pushed us to explore innovative solutions and to think critically about system design and user interaction.

Our project's success was quantified not just by the functionality of the game but also by the efficiency metrics such as logic utilization, power consumption, and operational stability. We achieved a balance between complexity and performance, ensuring that our Tetris game ran smoothly without consuming excessive power or resources.

In our Tetris implementation, we encountered a significant design challenge related to the use of latches, which inadvertently became part of our logic due to non-blocking assignments in combinational blocks. These latches introduced potential timing issues, particularly evident in the collision and rotation logic when interacting with the game via keyboard inputs. The presence of latches can lead to unpredictable behavior under certain conditions because their output depends not only on the current inputs but also on previous states, which are not always consistently updated across clock cycles. This resulted in sporadic responsiveness to control inputs, affecting the smoothness and predictability of block rotations and movements.