**ECE 350: Digital Systems**

**Final Technical Report**

Shriya Challam and Arnav Nayak

04/28/2024

**Overview: Overall Project Design and Specifications**

Tetris is a game created in 1985 that consists of colored blocks moving downwards in a grid. The goal of the game is to move the blocks left and right to get full rows. When a row is full, it will be cleared, allowing the player to continue placing more blocks. The game ends when the blocks reach past the top of the grid.

The goal of our final project was to create a simplified version of tetris. We aimed to handle all user inputs through the buttons on the FPGA board. We used the VGA display to show the main screen of the board. We also kept track of user score and current level.

****

**Figure 1: An early version of our Tetris game before we implemented score and levels.**

**Final Rubric Analysis**

Our final project met all of our desired specifications of our minimum viable product except for the 1. Our desired MVP included:

* Fixed game area of standard Tetris Grid
* A set of 3-5 tetris blocks that can move down the grid
* Line clearing when a row is completely filled
* Game over logic when blocks stack to the top of the grid
* Controls for moving the blocks

Our final product was able to complete all of these except line clearing. We also added a few extra elements including:

* Scoring based on added blocks and lines being fully filled
* 3 different levels that corresponded to different speeds of the block falling
* Level incrementing when a certain score is reached

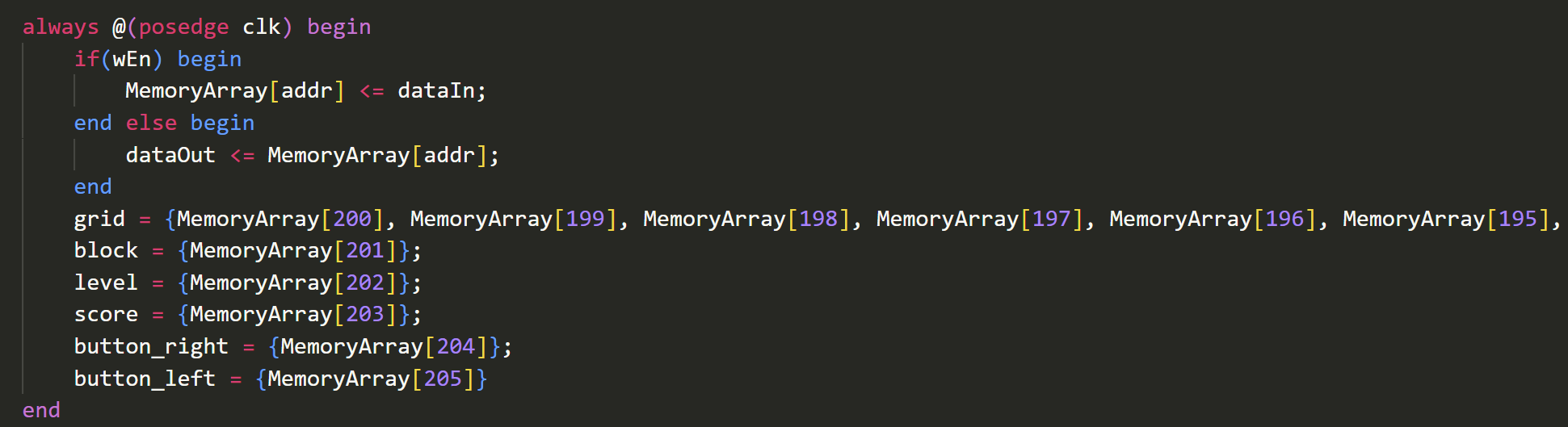
**Input and Output**

The input was through the FPGA board buttons. The player is able to control the falling blocks movement left and right using these buttons. These inputs would communicate with the core application logic in assembly through memory mapped I/O.

The output is the visuals on the VGA display. The VGA display shows the game area, current level, and current score. The current block will be falling on the grid and any button clicks by the player will move the falling block accordingly. The VGA rendered the state of the game by reading it from memory.

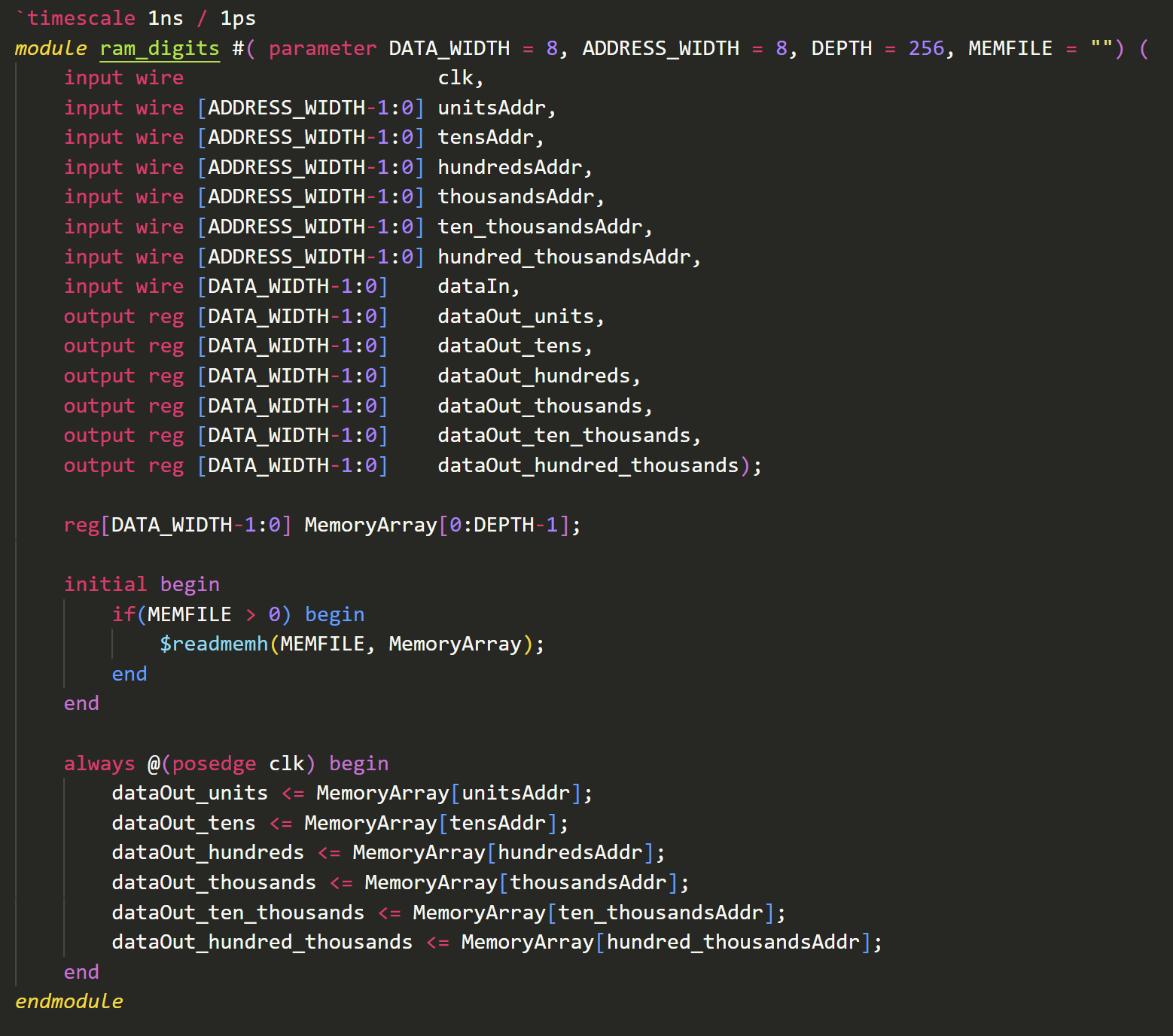
**Changes to Processor**

We did not make any drastic changes to our processor. All changes made to the processor allowed us to use memory mapped input/output faster. Our memory held the current state of the game. We used the first 200 (1-200) bits to hold the 20x10 grid cells linearly. A 1 meant the grid was occupied and a 0 meant it was not. We used the next bit (201) to hold the cell the current falling block was in. Bits 202 and 203 were used to hold the level and the current score. If a button was pressed, we passed that information to the RAM and stored it in bits 204 and 205, for the right and left bits respectively. This allowed us to continually check and update the memory through our assembly. Our CPU would then continually be reading from the memory and rendering the blocks, grid, level, and score accordingly.



**Figure 2: Part of our processor’s RAM where we stored the values mentioned above in the memory allowing for memory mapped input and output.**

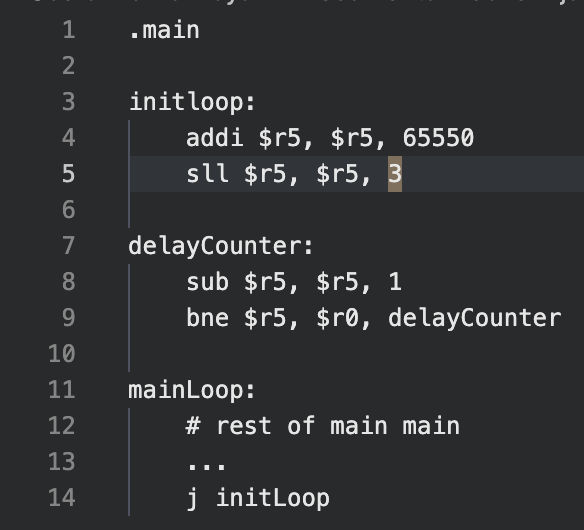
In order to display the score and level on our VGA screen, we created a custom Verilog module called binary\_converter. This module would take the binary score or level as the input and convert it to a decimal number. It would output the units, tens, hundreds, thousands, ten thousands, and hundred thousands digit. We modified the RAM that read in image memory files to take in multiple input addresses and output multiple sections of the memory file. Using this, we were able to render each digit of the score and level by converting to decimal and reading in the memory file once.



**Figure 3: Our VGA\_digits module that allows us to get the memory for each digit of a decimal number by only reading the memory file once.**

**Description of Assembly**

The entirety of our game logic lived in our assembly. Our MIPS code continually looped, read and performed operations on certain bits from memory. In Tetris, blocks fall cell by cell every ~1 second. We recreated this effect by forcing the loop to count to a large number (~10 million, bit shifted to load value into register) before performing calculations on the block’s position.



**Figure 4: Game Loop with Delay Counter Framework**

In each loop of the assembly, the current block position, represented by the cell index, was read in from memory. In order to move the block down, we would add the correct offset to the current position and place it back in memory. This would update a block to move down in the grid. For horizontal movement, we continually checked a specific position in memory for a toggled bit (toggled by the VGA on user button input), clearing it every time it was triggered. If the right or left button was pressed during that cycle, we would add 1 or -1 to the block’s position (ensuring there is no collision with an already occupied cell) before storing it back in memory, moving the block a cell to the right or left.

Before updating the block’s position to the newly calculated position, we performed collision checks. For each movement of the block, we ensured that the potential next position of the block was bounded by the grid, including the bottom most point in the grid. We also checked if the next position was already occupied by another block in the grid. If either of these were true, then a collision has occurred. The grid cell at the position of the block would be converted to an occupied space (toggling the memory position to 1). Finally, the block would loop back to the top of the grid and restart its falling process. This loop allowed us to only have to manage 1 block at a time.

The MIPS also handled updating score and level. Anytime a block hit the bottom of the grid, we loaded the current score from memory, updated it by 10 and stored it back in memory. Anytime the score reached 1000, we updated the level in the same way. The VGA handled rendering these values from memory.

We implemented pieces by duplicating our falling block logic in order to keep track of 4 blocks instead of 1. We implemented different shapes for the blocks by mapping a 4-but code to a type of block. The first 4 bits corresponded to the cell position of the block. The second 4-bits corresponded to the type of block. (0000 → T block, 0001 → L right block, 0010 → L right block, 0011 → square block, 0100 → long, straight block). The falling logic remained the same. When we performed our collision checks, we first checked the type of block and then pinpointed the rightmost and leftmost block in each piece (identified their positions in memory). We performed horizontal collision checking using those blocks as our reference instead. Collision checking with the grid bounds also remained the same.

The last part of the MIPS code included the end game logic which checked if an occupied block ever reached the top row. We do this by checking if a point of collision occurred on the first row. If so, the user has reached the top of the grid and the game ends.

**Challenges**

The biggest challenge that we faced throughout this process was understanding the relationship between the processor and the VGA controller. From the labs in class, we assumed it would be possible to control a majority of the game logic within the VGA controller. We originally planned to use the processor solely to keep track of score and leveling. However, this proved extremely difficult because we had to create multiple always loops that continually checked for any actions in the VGA controller. The game state got too muddled, and we realized we were not using the processor we built to its full potential. Switching the game logic to live within our assembly fixed a lot of these problems.

Another challenge we faced was with creating complex block pieces and backgrounds. When trying to create multiple scenes such as a start scene and a game over scene, we quickly ran into an error saying we ran out of BRAM. After asking multiple TAs, this problem seemed unavoidable, so we decided to not have multiple scenes. We still ran into the issue with the memory for the different blocks. We fixed this by converting all the colors to smaller bit colors to conserve BRAM.

**Description of Testing**

For our Verilog modules, we created test benches to verify their functionality. We ensured that each module generated the correct outputs. One module we had to create was a binary to decimal converter in order to convert our score to decimal for the VGA to render. After spending 2 hours debugging, we found out that it was because of a bug in our BCD module. We found out after writing a test bench and seeing an unexpected output.

For our MIPS code, we used the following online MIPS interpreter (<https://dannyqiu.me/mips-interpreter/>). This was an online tool that simulated MIPS commands, registers and small amounts of memory. Anytime we needed to test the functionality of our code, we would simulate it here. This method for testing was slightly annoying, however, because the interpreter used a slightly different set of inst

**Improvements and Future Features**

There are two main features that we want to add to this project. The first feature we would add is row clearing. When an entire row in the grid gets filled, we want to clear the row and shift all the occupied blocks down one grid position. We were extremely close to accomplishing this goal, but were not able to finish in time for our demo. In our assembly, we checked each row if each cell was occupied. We were able to increment the score accordingly if this happened. However, we struggled to shift all the rows above because our grid was stored linearly. Because we were dealing with this issue, we did not include this part of our assembly in our demo. After talking to Professor Board, he recommended we store our grid data in columns to allow for vertical shifting. In order to complete this task for our MVP, we would change our grid to be stored vertically, so shifting would be very intuitive.

The second feature we would add is rotating blocks. Upon clicking the up or down button, the block that is currently falling down should rotate 90 degrees either left or right. We would encode the rotation of a block in the third 4-bit section of the block data. For example, 0000 would correspond to an upright position and 0001 would correspond to a 90 degree right turn. Therefore, the memory that holds the current block information would hold the cell position, type of block, and orientation. There would be additional pointers if the button up or button down was pressed, similar to how the buttons left and right are handled, to indicate when to increment the rotation.