ECE532: Hardware Accelerated Tetris AI Group Report

Group 5
Kyeong Mo Kang
Derek Peterson
Ryan Xi

Table of Contents

GitHub repository: https://github.com/DerekPPeterson/ece532

1 Overview

1.1 Motivation

Tetris is one of the most played game since its release. As many other games, AIs capable of clearing millions of lines have been developed, and there are ongoing researches on making it better. However, most if not all of the AIs are written in software. The algorithm is highly parallelizable and uses logical operation, fitting for a digital circuit. By using parallelized hardware, the algorithm should run faster, and potentially can cover more cases using heuristic algorithm.

1.2 Goals

The goal of this project is to design and implement a Tetris AI in Hardware. A working Tetris that is able to run on Microblaze also needs to be implemented, and the game and AI must be able to communicate to each other to play Tetris. Once the game is started, the AI should be able to continue playing the game for as long as it can without any further input. The hardware should compute the optimal moves for the next piece faster than the software can.

1.3 System Change

Our original focus of the project was to incorporate and implement heuristic algorithm for what we hoped to have a better performance. However we realized during the AI implementation that the AI itself was quite a task to implement, and we could not get it to work on par with the software equivalent.
We focused more on trying to make the AI better and make the objective of the project to be hardware accelerated AI.

1.4 System Block Diagram

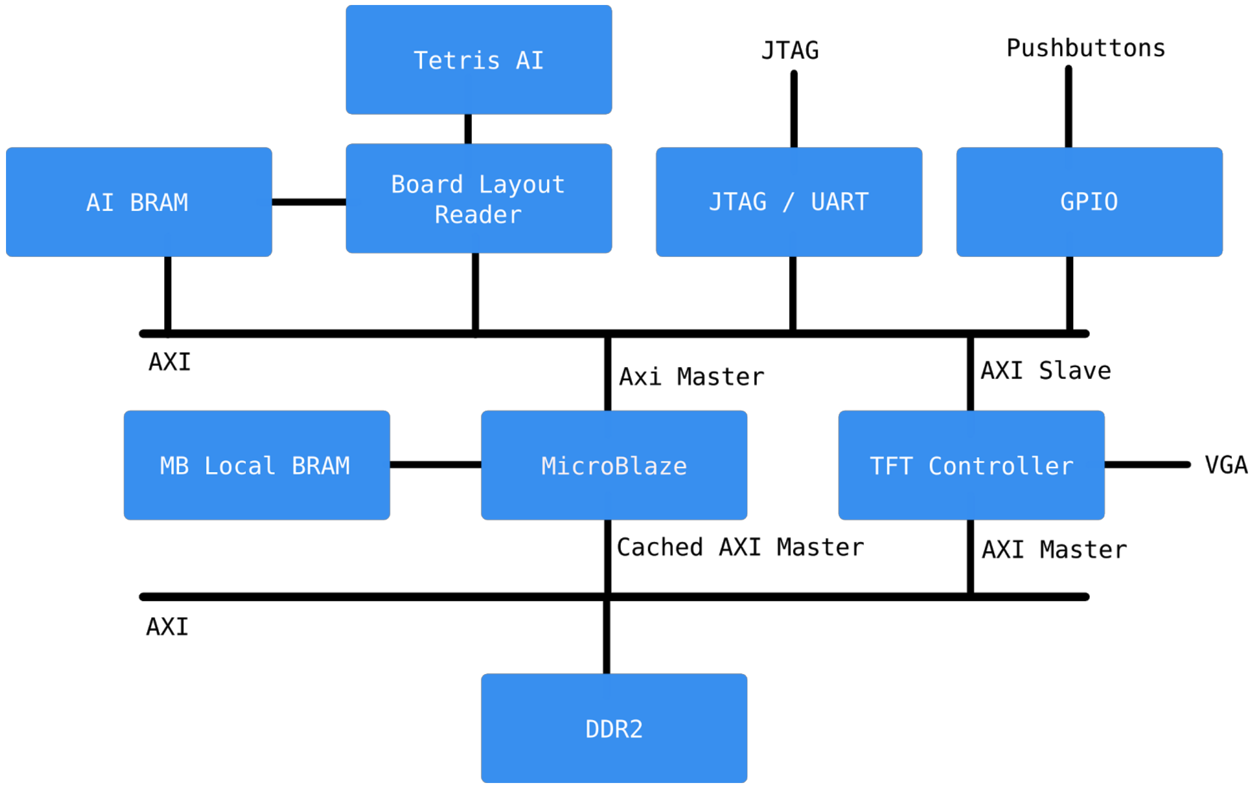

Figure 1.1: System Block Diagram

1.5 Brief Description of IPs

Table 1.1: IP Description

IP	Version	Description	Function
Provided IP			
microblaze	9.3	32-bit soft processor	Run Tetris program
AXI Interconnect	2.1	Communication path between microblaze and other IP	Access RAM, BRAM, TFT, and AI from Microblaze
AXI TFT Controller	2.0	VGA TFT Controller	Display tetris game on VGA monitor
AXI BRAM Controller	4.0	BRAM memory controller	Used by Tetris software to write board state and next piece ID to BRAM for tetris AI
Block memory Generator (AI BRAM)	8.2	Configurable Block RAM	Stores board state that tetris AI uses. True Dual Port used, allowing for both microblaze and AI access.

AXI Uartlite	2.0	Universal Asynchronous Receiver Transmitter used to communicate over RS232 Connection	Displays number of lines removed before game over, and running average
AXI GPIO	2.0	AXI interface to general purpose IO available on board	When AI not playing, human input done through GPIO.
Memory Interface Generator	7.3	DDR Ram memory controller	Used by Tetris software to build FIFO for VGA display
Custom IP			
Board Layout Reader	1.1	AXI Peripheral interface between Tetris AI IP and Microblaze	Reads board state from BRAM, and outputs it with enable signal every new piece. Then forwards, new piece placement from tetris AI IP to Microblaze with done signal.
Tetris AI	2.5	Calculates best piece placement for tetris given the current board state and next piece.	Decides the optimal placement of the next piece for tetris software

2 Outcome

2.1 Results

The system was able to play the game to relative success. Figure 2.1 and table 2.1 shows the statistic of two of AI we created. The "Older" AI had the best performance, with average of approximately 4000 lines cleared. The "Newest" AI tried to mimic the software equivalent completely. However, there was at least one bug that deviates the score from the value software calculates, and was only able to clear average of 1000 lines.

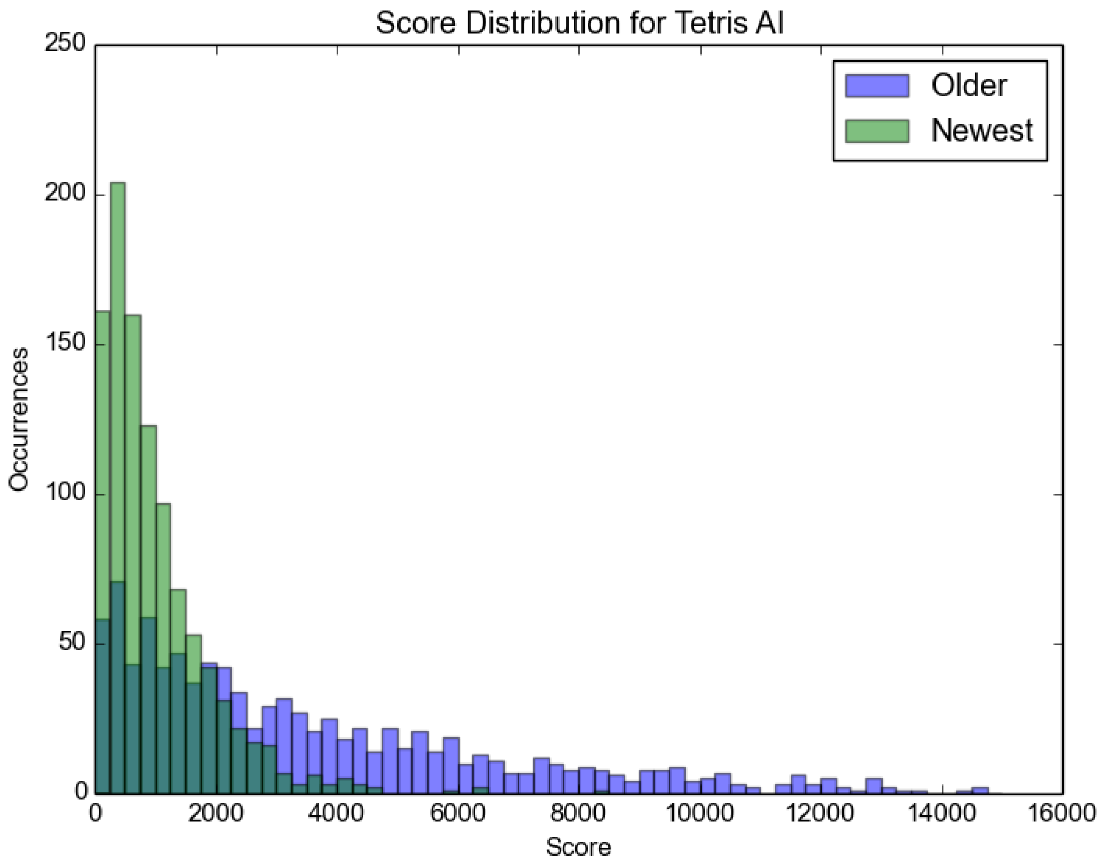

Figure 2.1: Distribution graph for number of lines cleared using two AIs

Table 2.1: Numerical Statistic of AI - values in number of lines cleared

Older	Newer
count: 990 max: 34803 (not shown) avg: 3998 median: 2723.0	count: 1027 max: 8301 avg: 988 median: 731.0

The hardware was also able to compute and determine the optimal placement of the pieces in a relatively short time. It's difficult to determine how fast the software AI runs, but by timing the speed El Tetris [1] and looking at a few online posts[2], it seems like it can run at approximately 10000 pieces per second, or 100us per piece. The designed hardware can compute the optimal placement in 156 clock cycles, which is 1.56us per piece when 100MHz clock is used. The number of cycles can even be reduced to 91 cycles if more LUT was available.
However, the hardware seemed to run slower. We assumed this was due to the slower clock speed in Microblaze running the Tetris software which was causing a bottleneck for the apparent output speed.

2.2 Proposed Improvements

The system was able to play the game to a relative success. However, the software equivalent we tried to implement clears average of two million lines when it's ran whereas the hardware only runs for a few thousand. That definitely should be improved to match the effectiveness of the algorithm.
Implementing heuristic algorithm is another improvement that may increase the number of cleared line significantly. Many of the game over state showed that the AI was not capable of handling board state where two columns are empty as it usually tries its best not to create a single empty column and simply stacks the pieces higher and higher. However, by using heuristic of two or three it should be able to realize the empty column can easily be removed a piece or two later.

3 Project Schedule

Table 3.1 Original schedule

	Week 1	Week 2	Week 3	Week 4	Week 5	Week 6
Derek	Figure out how Tetris input will be provided (Wrote own Tetris program)	Create system that runs the Tetris program, debug	Write software to control the AI hardware block	Software to support greater depth searches accelerated by hardware block / assistance with other parts	Integrate	Debug

Kris	Research AI algorithm Design Corresponding hardware block	Hardware implementation of BRAM and AI controller IP	Integrate system and debug.	Lower timing delay of controller block and increase controller ports for AI depth of 2-4.	Integrate	Debug
Ryan	Research AI algorithm Design corresponding hardware block	Write Verilog for the AI block. Simulate and debug.	Finish debug, integrate with system. Design heuristic algo.	Write Verilog or modify AI block accordingly to incorporate heuristic algo	Integrate	Debug

Table 3.1 shows the original planned schedule. We followed plan for the first two weeks, then the schedule started pushing. The third week was entirely spent debugging, and fourth integrating. We had a fundamental protocol issue where different ordering of IDs were being used which inhibited progress for some time. After successful integration, we realized that the hardware AI is not nearly as good as the software equivalent, so we decided to go through multiple cycle of designing, debugging and simulating in order to improve the AI. This was done through week 5 and 6, and we were able to improve the result by about 4 times compared to the first integrated version.

4 Description of the Blocks

4.1 Microblaze

The MicroBlaze is the processor IP provided by Xilinx that was used to run the software controlling the tetris game. It was configured to use a 64kB BRAM as it’s instruction memory. It was configured with a barrel shifter, integer multiplier and divider as well as a branch target cache to improve performance. To improve write speed to the DDR for the video buffers, a cached AXI data interface was used. Other peripherals used a normal axi bus.

4.2 AXI BRAM/AI BRAM

BRAM used in our design was obtained from the IP library provided by Xilinx. Then, it was configured to True Dual Port RAM memory type in IP configuration. This type was used over the default Single Port RAM, since we wanted the Tetris AI to be able to read or write from the BRAM without having to interface through the AXI. Figure 4.1 shows the BRAM connection with the board layout reader. The blue line from BRAM_PORTA is a AXI connection going to the Interconnect. On the other hand, the ports in BRAM_PORTB connect to Board Layout Reader.

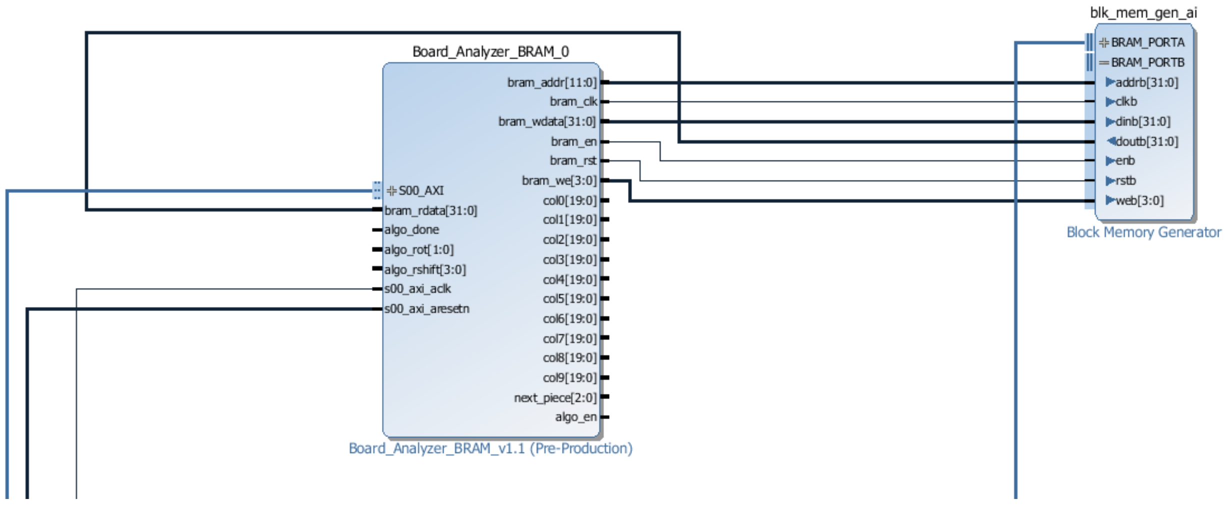

Figure 4.1: AI BRAM connection to AXI and Board Layout Reader

4.3 Board Layout Reader

The main functionality of the board layout reader was to read the board state stored in the BRAM by the tetris software. It was decided to use an AXI interface IP for this. By doing that, the board layout reader could be used as the communication interface between tetris and the AI. Three slave registers are utilized in the board layout reader. After tetris software has finished placing the current board state in the BRAM, it stores BRAM address into slave register 0. Following that, it raises the slave register 1’s zero bit, signalling the board layout reader to start loading data from the BRAM. Figure 4.2 shows the control signals used to do this. This waveform was acquired from BRAM memory controller specification sheet.

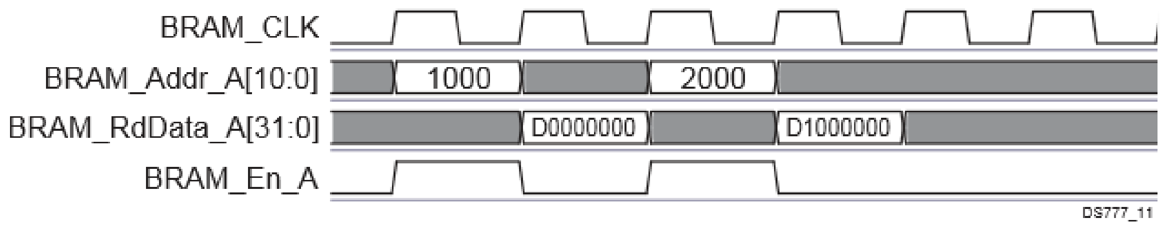

Figure 4.2: BRAM control signal

After all the datas are loaded and stored onto the column registers ranging from 0 to 9, and the next_piece ID, the algo_en signal shown in Figure 4.3 is raised. This signals the Tetris AI to start evaluating the best placement of the next piece given the tetris board. Then the board layout reader waits until tetris AI raises the algo_done signal before storing algo_rot and algo_rshift into slave register2, along with finish signal for the tetris to read.

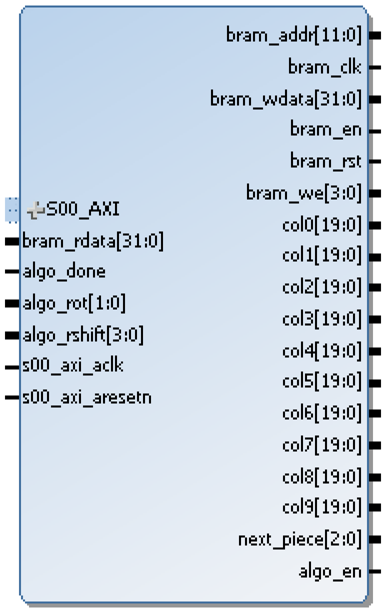

Figure 4.3: Input and outputs of bram layout reader

4.4 Tetris AI

The tetris AI IP ports are shown in Figure 4.4.

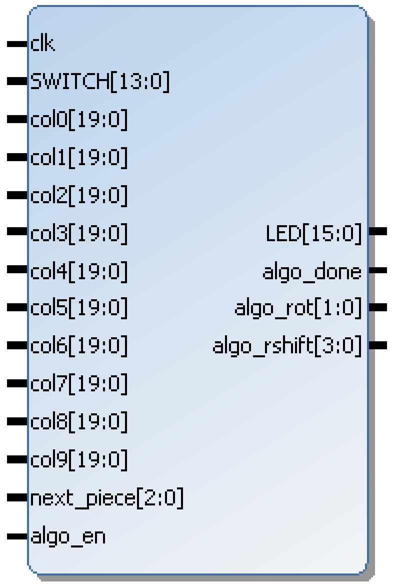

Figure 4.4: Tetris AI ports. Left side ports are inputs, and right side ports are outputs

The IP takes input of column 0 to column 9 registers, next_piece ID and algo enable. All of these inputs come from the Board Layout Reader IP. When all the inputs are set and algo enable is raised, the IP uses hardwired next_piece ID settings to determine how many rotations and column placements the new piece can do in the tetris board. Then with these settings, the next piece placement logic and evaluation logic are ran for each of the possible placements. This is visualized in Figure 4.5. For example, I piece has two unique rotation, and 10 unique column placements. On the other hand, T piece has 4 unique rotation, but either 7 or 8 column placements depending on the rotation. During the placement and evaluation phase, all those placements and evaluations are done in parallel, by creating multiple instantiations of the said logic.

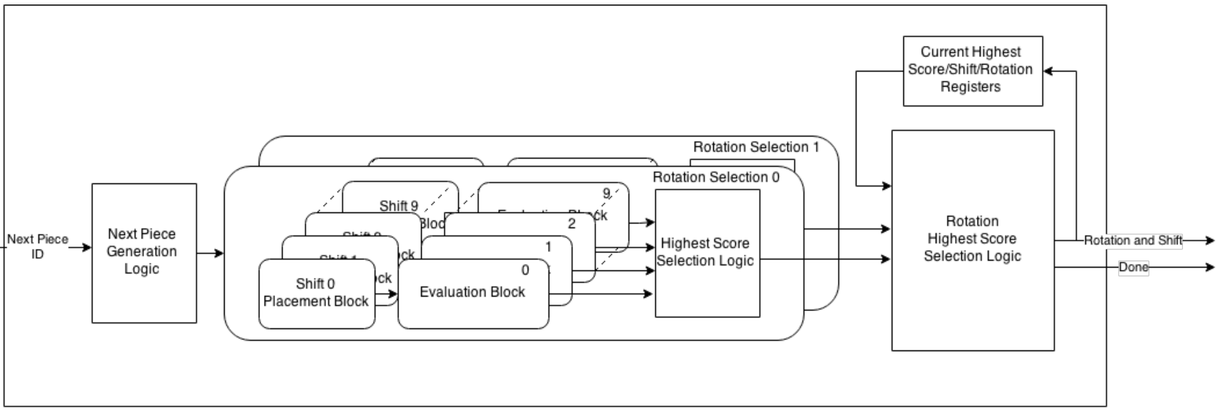
Next Piece ID
Next Piece Generation Logic
Rotation Selection 1
Rotation Selection 0
Shift 9
Shift 0 Placement Block
Evaluation Block
0
1
2
9
Highest Score Selection Logic
Current Highest Score/Shift/Rotation Registers
Rotation Highest Score Selection Logic
Rotation and Shift
Done

Figure 4.5: Block diagram of the Tetris AI logic

One thing to note though is that due to lack of LUT area, not all 40 of the possible placements could be implemented into the FPGA. As a result, only two instantiations were implemented, and they were used in series of two rotations followed by the next two. Statistically, this slows down the Tetris AI by only 37%. That is because out of the 8 pieces, only T, L and J has unique rotations greater than 2. In following subsections, the details of the placement and evaluation logic will be discussed.

4.4.1 Placement

The placement logic takes in 4x4 registers representing the next piece, as well as 4x20 columns for placement. Placement of next piece is done by calculating the AND of the next piece registers and the columns top down until there any of the 4x4 output is 1. At this point, the next piece is shifted back up and placed into the columns with the use of OR logic, and stored in output 4x20 registers. Error cases are handled as well for when the next piece cannot be placed into the 4 columns. For example, the columns could be filled up to 18th or 19th row. In that case, I would now be able to be placed, and evaluation logic will not be ran.

4.4.2 Evaluation

The evaluation block is a FSM controlled array of 2D two directional shift registers. By performing a complete circular shift horizontally and vertically once each, it counts the number of features used to calculate the board state score. This block is a black box to the rest of the system, meaning as long as it receives appropriate input, it will simply output a score value.

The scoring scheme was implemented as described in El Tetris software AI. [3] There are six features it tracks described below:

- **Landing Height**: Original height of the column placed + piece height
- **Rows removed**: Number of rows filled and removed
- **Row Transitions**: Number of two horizontally adjacent cells where one is filled and other is not
- **Column Transitions**: Same as row transitions, but vertical.
- **Number of holes**: Number of empty cells where at least one cell above in the same column is filled
- **Number of Wells**: Number of empty cells, where no cell above in the same column is filled, and both horizontal adjacent cell is either filled or is against the side wall

Each of the feature is counted and multiplied against a weight to calculate the final score. The weight is also taken from El Tetris. However, the original weight is in floating point number but we modified them to use integer. This due to the large area and resource usage a floating point multiplier would take up on the FPGA and we did not have the luxury of letting the score calculation module to take up so much. Below are the weight used for each of the feature.

Landing Height:	-4500
Rows Removed:	+3418
Row Transitions:	-3217
Column Transitions:	-9348
Number of Holes:	-7899
Number of Wells:	-3385

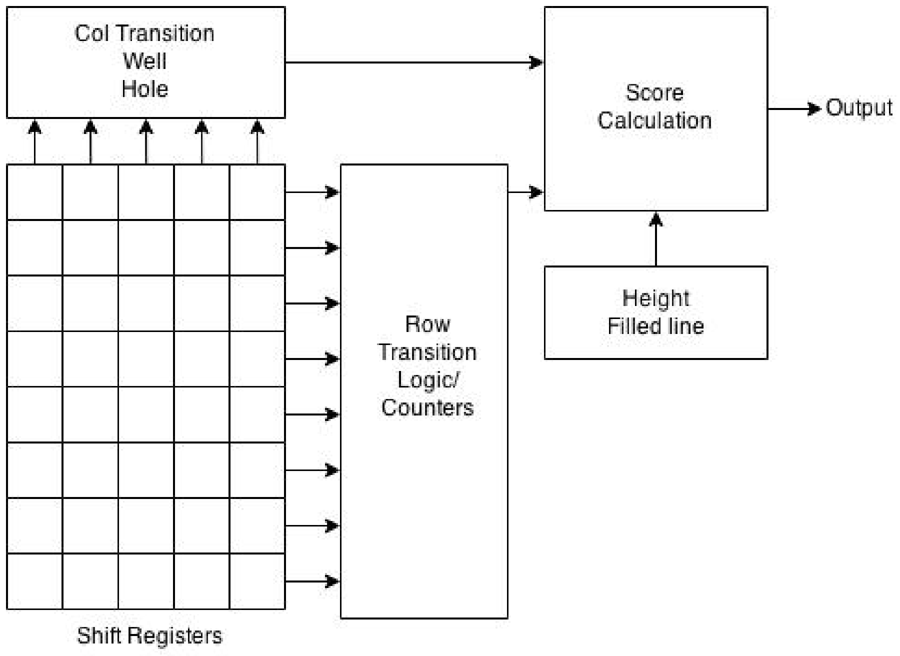

Figure 4.6: simplified block diagram of the module.

- The module receives ten 20 bit value as board state, along with go and active low reset signal as input
- a bit-wise AND on each row detects any filled lines
- In newer version, it goes through 20 shift ups to remove the filled lines. The best and used version skips this and the filled lines are taken into account logically when counting each feature.
- Rotate shift up 20 times and between each shift, logics attached to the upper two rows of registers counts column transition, well, holes and height. After 20 shifts, the board state returns to the original states as there are 20 rows total.
- Rotate shift right 10 times and between each shift, logics attached to the right two columns of registers counts row transitions. After 10 shifts, the board state returns to the original state as there are 10 columns total.
- There is an adjustment state where in one cycle, it increments or decrements appropriate counters if needed, for edge cases. For example, because shift rotate is used, there is a time where the right two columns are column 0 and 9 in the case of horizontal shift, and it should not be counted for row transition.
- All of the counters are summed and score is calculated

4.5 TFT Controller

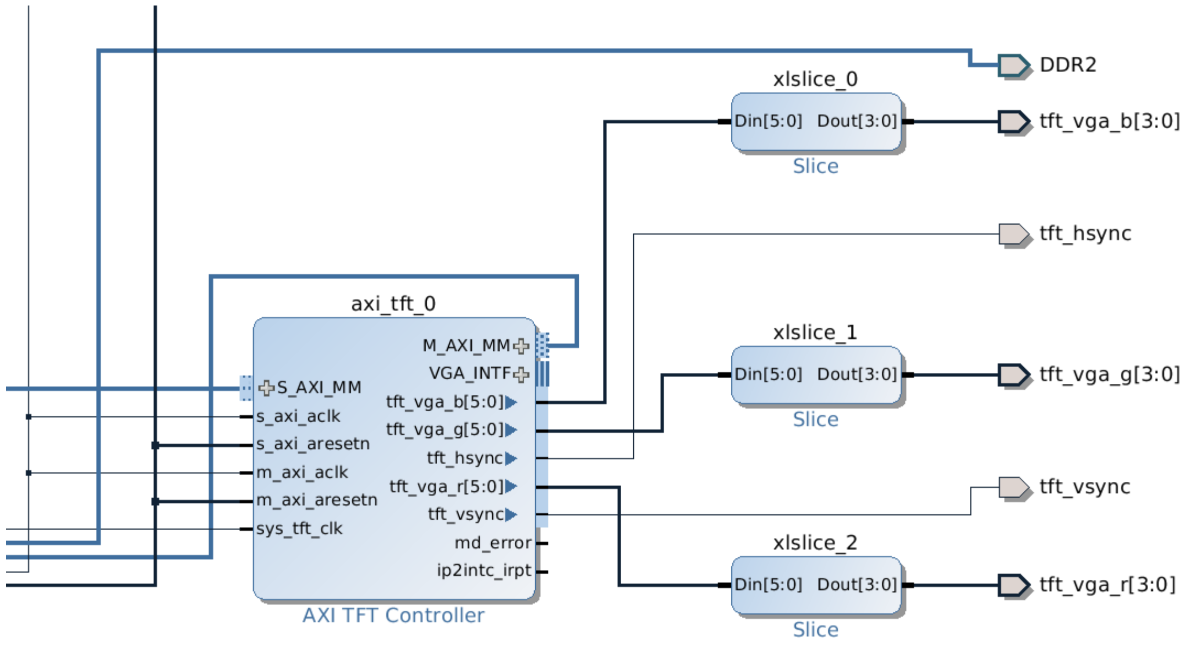

Figure 4.7: TFT controller diagram

The TFT controller was used to display the data in the video buffers on the vga display. The TFT controller is connected on the normal axi bus as a slave so that the Microblaze can access its control registers, and it is on the axi bus connected to the DDR as a master so that it can access the video buffers.

The slice blocks are used because the TFT controller supports 6-bits for each colour, but the Nexys4 board only uses 4-bits per colour. The slices are needed to connect only the lower order bits.

5 Description of Design Tree

Github documentation is located at: https://github.com/DerekPPeterson/ece532
The final system design is in designs/tetris_test2. The IPs are located in designs/ip_repo.
In the ip_repo, aicontroller holds the Tetris AI IP, and the board_analyzer_bram_v1.0 holds the Board Layout Reader. For the Tetris AI IP. Look at the following README.md for details.

README.md

Project Name: Hardware Accelerated Tetris AI

Description: FPGA is used to calculate the optimal next piece placement for a software tetris running on Microblaze. The purpose was to see if the hardware accelerated AI will be able to perform as well as software AI.

How to Use:
1) Download the repository
2) Open designs/tetris_test2/tetris_test2.xpr on Vivado.
3) Open project Settings and add the ip_repo into the ip repository
4) Save settings and open block diagram to ensure that none of the blocks are missing
5) Synthesize, implement and generate bitstream.
6) Export to SDK, program and enjoy.

Authors: Derek Peterson, Ryan Xi, and Kyeong Mo Kang.

Acknowledgements: Thank you Professor for taking time to guide us weekly.

6 Tips and Tricks

- Standardize interface protocol between blocks at the very beginning of the design phase.
- Writing to DDR is significantly faster when using a cached AXI inferface
  - Useful if you are updating the screen using software rather than hardware
- Some inexplicable software bugs can be caused by an incorrect linker script that places portions in the code in undesirable locations (e.g. placing code in the video buffer!) Change the linker script settings by right clicking on the project in the SDK and selecting Generate Linker Script

Reference

[1]Islam El-Ashi. “El-Tetris in HTML5”. Aug 13, 2011. Web.
http://ielashi.com/el-tetris-in-html5/
[2]”Calculating holes on a tetris grid quickly”. Feb 13, 2013. Web
http://codereview.stackexchange.com/questions/80462/calculating-holes-on-a-tetris-grid-quick
ly
[3]Islam El- Aish. “El-Tetris - an improvement on pierre dellacheries algorithm”. Jun 1, 2011.
Web. http://ielashi.com/el-tetris-an-improvement-on-pierre-dellacheries-algorithm/