EECS 314 – Computer Architecture

MIPS Tetris

Final Report

Umang Banugaria, John Gunderman, Nathan Starr, Matthew Wallerman
5/4/2012

Problem Statement

Our group decided to create an implementation of Tetris, a classic tile matching puzzle game, using the MIPS language and run through SPIM. Different shaped blocks fall to the bottom of the screen in a random sequence, and the player has the ability to move the blocks. Another block only falls after the first has hit the bottom of the screen or the block hits another and stacks upon the older block. The player accumulates points when a whole row of blocks is created, and when this happens the row is cleared and all blocks above are shifted down to fill the missing row. After a certain amount of points the rate at which the blocks fall increases. The player loses if the blocks reach the top of the screen and another block cannot fall.

Major Challenges

There were numerous issues that we had to address when implementing our game. The first issue we needed to tackle was the fact that there is no graphics processing in MIPS. This is quite a big obstacle when making a game. We needed to figure out a way to implement graphics in MIPS. Another issue related to graphics processing is the need for a timer. In order to process the falling of the blocks we needed to implement a timing mechanism.

A challenge directly related to MIPS that we needed to overcome was the fact there is no abstraction in MIPS. What that meant for us was that every block needs to have its own procedure dictating its shape. Not only does it need this, but every piece's orientation and movement while at that orientation needs to have its own procedure. This meant that we would need large amounts of code to produce each piece. We run into a similar problem when trying to handle the game logic. This includes clearing rows and ending the game when a column gets stacked to completion.

The set and get methods helped abstract the board some levels but it did not help us deal with the pieces. We could not come up with a way to make piece creation and manipulation eloquent so we had to do it the hard way. Each piece has its own main procedure that gets jumped to when that piece is created. Once we are inside the procedure of a piece the game is handled there until that piece is finished and we move on to create a new one. Each piece has a series of sub-procedures depending on how complicated that piece. Each sub-procedure related to a piece contains different shift-left, shift-right, drop, and rotate procedures for each orientation that that piece can have. The simplest piece was the square since it has no rotations while the most complicated pieces were the L, backwards-L, and T pieces.

Key Components

The first major component of our program is the top level graphics processing. We accomplished the need for graphics and a timer by using the Python library, Pygame. Pygame lets us draw the blocks to our screen as well as implement a simple timer to control each game event.Pygame reads in our board state and determines where to place each colored block. We are also able to implement our timer in Pygame by using Python's built in timer library.

The biggest component of our project on the MIPS side of things was handling the board object. Since MIPS does not have a natural way to build two-dimensional arrays we chose to represent our board using a single one-dimensional array. Since this drastically increases the difficulty in creating and moving pieces around our board we decided to build a coordinate system to make navigating the board much

easier. The most natural way to do this was to create X and Y coordinates to represent each space on the board with (0,0) being the top right corner of the board. To make these more functional, we wrote get and set methods for our board that could take in an X and Y coordinate and store a particular value there. This made moving our pieces around much simpler since we only had to manipulate X and Y values instead of trying to figure out which places in our single array that needed to be manipulated.

Integration of Components

We had Pygame displaying the graphics of the board and MIPS was handling all of the logic of the game but we needed a way to communicate between Python and MIPS. We were able to accomplish this by using Pythons ability to launch subprocesses. Python creates a subprocess which launches SPIM and hijacks STDOUT, STDIN, and STDERR and redirects them to the pipe. This way, all MIPS has to do is use a syscall to print to the screen and since MIPS prints through STDOUT, Python can read all of the information being printed just like it was reading a file. In a similar fashion, when we want to pass information from Python back into MIPS, we have MIPS wait on integer input and then Python writes directly to STDIN the value it wants MIPS to receive as well as a newline character to simulate the enter key being pressed.

We used a numeric system to handle all of the communication between MIPS and Python. Once the program is run MIPS will create an initial board state of all zeros. The board state was represented by sequence of 128 digits. The digits that appear in the board state are 0-7. To MIPS each digit indicates which piece is located at that position but to Python each number only represents color. This is to further detach Python and MIPS to show that all of the logic is really happening in MIPS. Python has no understanding of the pieces of Tetris, it only knows which color goes with which digit.

Once Python receives the initial board from MIPS it sits and waits for MIPS to prompt it. If Python receives an 8 from MIPS that means MIPS is waiting to drop a new piece. We then use Python's random number generator to generate a random integer 1-7 to send to MIPS as the next piece. MIPS receives this integer, determines which piece it represents and then goes to that pieces procedure. MIPS stays within that piece's procedure until the piece makes a collision and is done falling. At each tick of the clock, MIPS sends a 1 to Python which tells Python to send MIPS which key was hit last by the user. MIPS can't interpret the arrow keys on the keyboard so Python sends MIPS 1 if the left arrow key was hit, 2 if the right arrow key was hit, 3 if the up arrow key was hit, and 4 if no keys were hit. MIPS takes this integer and determines how to handle the input, makes the appropriate changes to the board state and then sends it back to Python in order to be displayed.

User Interface

The program is very simple to get running. All you need to do is to be on a system that is running unix and has "spim" and "Python-Pygame" installed. Then you navigate to the directory where our Python and MIPS scripts are located and run "python boardinterpreter.py." This will open a new window which will contain the game screen. To play the game you use the left and right arrow keys to shift the blocks and the up arrow key to rotate the blocks. To quit the game you simply close out of the window and Python will kill the instance of SPIM that it started.

“Heisenbug”

There is currently one bug in our program that we are aware of, however, we have yet to discover a way of addressing the bug or even to replicate the bug consistently. The bug occurs when the user tries to rotate the T-piece while it is near the top of the board. In some instances, rotating the T-piece near the top of the board will cause the piece to detect a collision and request a new piece to be sent even though there is nothing but open space below the piece. This bug does not occur with every instance of the T-piece falling. This has made this bug nearly impossible for us to debug because, after reviewing our code and performing multiple test runs, we have been unable to determine the cause of the bug or able to replicate it consistently. We have been able to lower the probability of the bug but have been unable to complete remove it. The bug has thus far never appeared when the T-piece has been falling without rotating. One theory of why this bug occurs is that the register we use to store the rotation state of the piece is somehow being overwritten making MIPS think that the piece is in a different orientation than it is supposed to be. However, this doesn’t explain why the bug doesn’t appear consistently nor does it explain why this bug only appears with the T-piece and no other pieces.

The Future

Ideas that we tossed around for continued work on this project involved having different difficulty levels, scoring, implementation of the down arrow, and preview next piece. The implementation of the down arrow would be the simplest thing to implement since we would handle that in Pygame by speeding up the game clock. Implementing preview piece would also be relatively simple since we would likely just launch a new SPIM process and have a new MIPS program which took the integer form Python and converted it to that piece to feed back into Python and be represented in a different window then our main game. Scoring would also be an easy implementation because we would only need to have a memory location that held the score that we would increment by ten with each line cleared. The hardest new feature to implement would be the increased levels since we would have to design what changes should be made to the game to increase the difficulty at a steady level. Unfortunately, we ran out of time while working on our project before we could see any of these ideas bloom into fruition.