Patrick Chiu
Mai-Anh Duong
David Fu
Edward Liu
18-545
Final Project Report

**Network Tetris on FPGA**

Tetris is a classic puzzle game where the object of the game is to manipulate tetrominoes to stack and fit together along a horizontal line. Tetrominoes are shapes made up of four blocks that fall down the playing board in a random series. When a horizontal line, without any gaps is created, the line disappears and any blocks above that line will shift down the playing field. The game ends when a player's stack of tetrominoes reaches the top of the playing field and new tetrominoes can not enter.

There are seven tetrominoes that the player can rotate and translate as they fall down the playing field. They are commonly referred to as I, J, L, O, S, T, and Z (shown below). The I tetromino is the only piece with the ability to simultaneously delete four lines. This action is called a tetris.

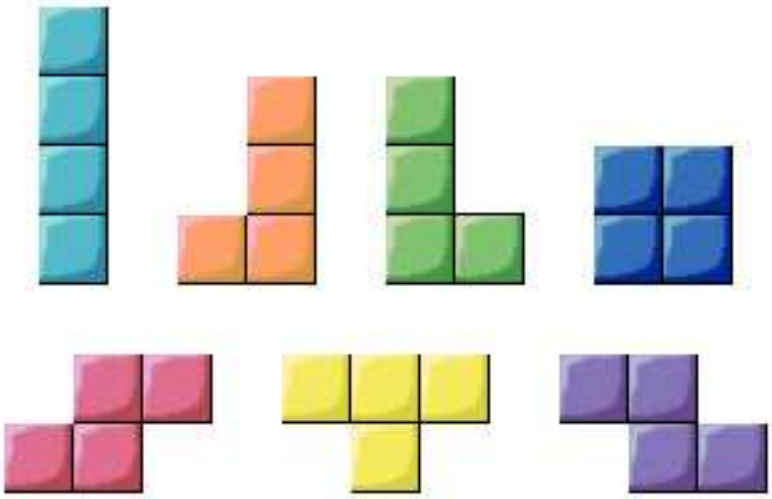

Figure 1: Tetromines in order of I, J, L, O, S, T, Z

In multiplayer tetris games, the winner is determined by the player who survives the longest. By deleting multiple lines at a time, a players can send garbage lines to their opponents. The garbage lines appear at the bottom of the board, causing the playing stack to shift up the field. The number of lines deleted and the corresponding number of lines added is shown below (Figure 2).

| Sending Lines | |
|---|---|
| # of Lines Deleted | # of Lines Generated |
| 2 | 1 |
| 3 | 2 |
| 4 | 4 |

Figure 2: Sending lines table

The goal of our project is to implement a single-player and multi-player tetris game in hardware on the FPGA. Each Xilinx board is programmed to support one to two player tetris games. By networking two Xilinx boards, a four person multiplayer game or four simultaneous single player games can be played. Software is used to transfer video and audio files from the flash drive to the DRAM. Game play, image output, and peripheral connections are implemented in hardware. Also, audio output and networking, via the Ethernet, are done in software.

**Hardware Implementation**

The single-player and multiplayer game code is completely done in hardware. The multiplayer game code is comprised of multiple single-player FSM's that are hard coded to communicate directly to each other. Each Xilinx board supports two single-player FSM's.

Random Number Generator

The random module used by this project is a pseudo-random number generator capable of generating a 32 bit random value. It uses a 52 bit LFSR. The code used for this module is from http://www.birdcomputer.ca/Cores/lib/rand.v.

This module is used by the game logic for two purposes. The first one is for deciding which piece will be spawned next and the second one is for generating the random lines added to an opponent's field on row clears.

Board Ram

This module holds the board state and is connected to both the game logic and the vga controller. The game logic updates this ram while the vga controller continuously reads this information. The module was coded in a way such that a dual ported block ram is inferred during synthesis.

Get Display & Get Next

These two modules function as our roms which held the information for a particular piece. Given a piece and rotation, it returns the information for the four blocks of the piece, width, height, color, and the next piece display information. The module was coded in a way such that a rom is inferred during synthesis.

Game Core

This module serves as the game core logic. It is connected to the board ram, get piece, get display, ps/2 keyboard interface, network signals, and another game core.

Using these input connections, the state machine implements our version of a networked multiplayer tetris.

In total, three different versions were produced. The initial version used up 50% of the board's resources clocking at a maximum frequency of 25MHz. The second version used up 40% but can clock 200MHz. Our final design uses up 6% of the board's resources and can clock at a maximum frequency of greater than 100MHz, allowing us to fit two cores on a board comfortably on the fpga fabric. The decrease in resource usage is mainly attributed to a reduction of barrel shifters which also caused shorter critical paths to increase our maximum clock speed.

On startup, the game core waits for a start signal from a user. It then proceeds to the main game flow with different behaviors dependent on the mode chosen. It firsts adds a piece, places it on the board, and checks for end game or landed/clear rows conditions. If it's valid, the game goes to the poll state which waits either for a timer tick or a user command. On a valid move or timer tick, the new location of the piece would be checked again for endgame/landed/clear row conditions. The behavior is dependent on the current mode.

Lines are added to opponents through asserting signals that are connected to either the other game core on the board or through the network with use of the software registers. The protocol for sending and receiving lines involves polling signals for differentiating tags with last handled tag. For example, if I want to send four lines across the network, I would increase my sent line tag by one, change the number of lines to send to four, and change the random send value which determines the lines being added. The receiving end would now see that the incoming tag is different than its last handled tag value and would proceed to process the add lines. New lines are only added when the board is in a committed state (when a piece has landed).

A similar system is used for telling the software to play certain sounds. On certain game events, the game core logic increments a tag that is stored in a software register to indicate to the audio logic that a new sound wants to be either played or stopped. The actual command and sound selected are transmitted along with the tag in software registers.

For an overview of the finite state machine state transitions, please refer to the attached diagram (figure 3).

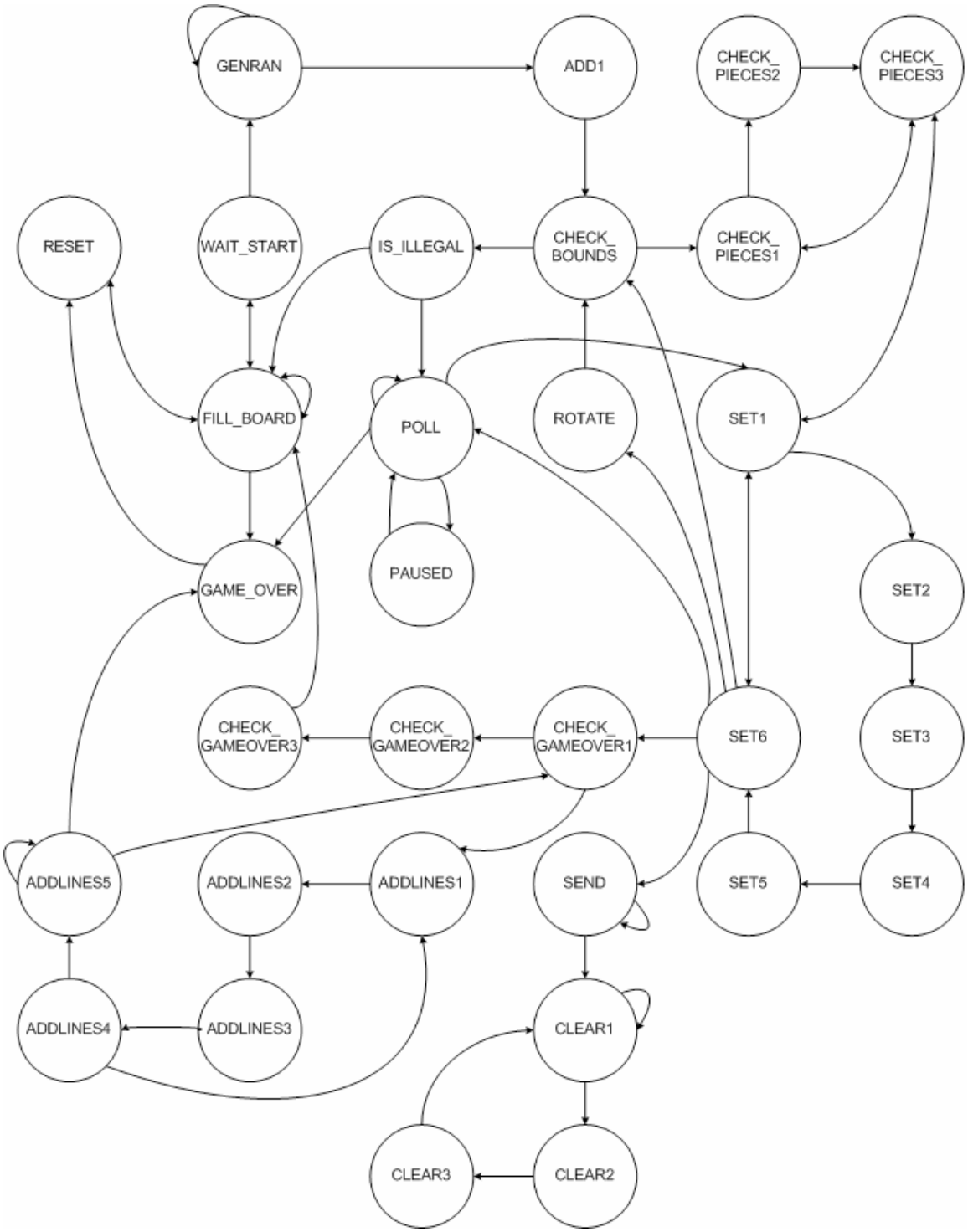


Figure 3: Single-player FSM

Top Level

This module connects all the vital components together.

1) Routes correct signals to the slave registers
2) Connects the proper network signal to each game core
3) Connects the keyboard signals to each game core

4) Connects the two game cores together
5) Holds the board rams for the game cores and vga controller

In a multiplayer game, there are two single-player FSM's that communicate directly to each other. The information sent between the FSM contains game state (ie. start, pause, end) and addline commands. Each player has its own keyboard interface, board and audio commands, and network communication.

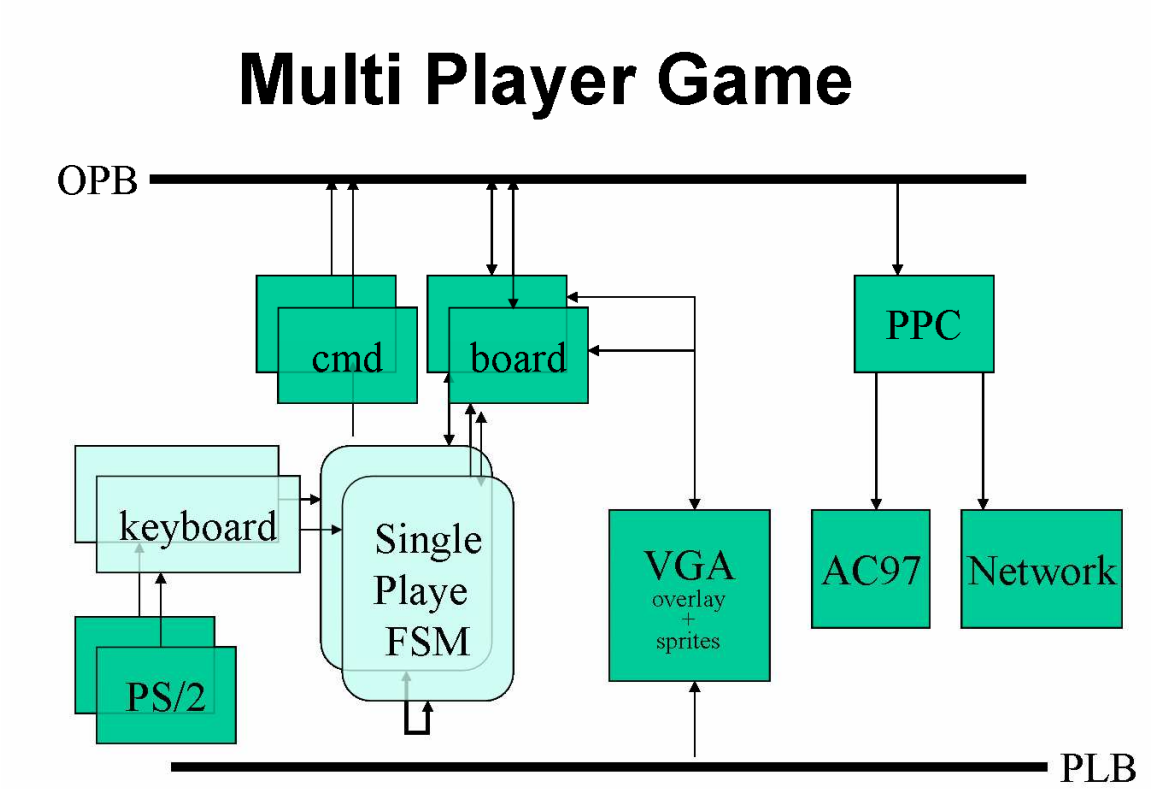


Figure 4: Multiplayer block diagram

PS/2 Keyboard Interface

The PS/2 keyboard interface module used by this project is originally from opencores http://www.opencores.org/cvsweb.shtml/ps2/ps2_keyboard.v. There was a slight modification of the code. It was changed from a bidirectional interface to a unidirectional where it only reads inputs from the keyboard. The reason for this simplification was problems encountered in integration of pull-up resistors with the bidirectional clock signals.

Two of these modules were instantiated, one for each PS/2 port.
The keys used are as follows:
F1: *single player mode
F2: *one versus one mode
F3: *network (one player vs network)
F4 *network (two players vs network)

F5: master mode (on network)
F6: user mode (on network)

Left-Arrow: Move Left
Right-Arrow: Move Right
Up-Arrow: Rotate Counter Clockwise
Down-Arrow: Move Down
Spacebar: Drop
ESC: Starts/Pauses Game

* means the key is only valid before a game starts

Video in hardware

The vga framebuffer given to us implemented the framebuffer in the DRAM and used a BRAM to buffer one line of the video output. The C code would use a setPixel command to tell the framebuffer on DRAM to change color. This utilized the PLB bus and the DCR bus to request a write to DRAM and to identify where the framebuffer was on the DRAM. We decided to remove the ppc portion of the vga framebuffer and directly overlay our changes onto our background. This would do the following:

1. Remove the unnecessary DCR bus and associated code
2. Remove write commands on the PLB due to the framebuffer
3. Relieve the ppc from the many loops needed to write a sprite into DRAM
4. Increase the resolution of our game

The first thing that had to be done was finding the specs for 1024x768 resolution. We required a 65Mhz pixel clock and different sync, front porch, back porch and active times. To get the 65Mhz pixel clock we inserted a new dcm module into the EDK and used the ClockFX port and divided the input 100Mhz signal by 20 and multiplied it by 13. We next added the ports required into the .ucf and updated our .mhs files. We rewrote the counter and timers for the vsync and hsync and tested them using modelSim. After a reasonable amount of testing, testing an entire screen would've taken too long, we additionally hard coded patterns to detect the correctness of our counters and syncs. We utilized checkerboard and striped patterns to verify that our counters were indeed correct.

After the syncs were verified working we noticed that the original vga framebuffer started requesting data from DRAM via the PLB when the vsync and hsync signals were in the active region, the region that is actually displayed. We decided that this would not be suitable for us because we needed to request ~4 times the amount of data that the original framebuffer required so using the same idea of having a BRAM buffer one line of the display we started requesting data from the PLB and writing it to our BRAM buffer at the start of the hsync sync signal. Since the order for the hsync and vsync signal is sync, back porch, active, front porch and back porch again we would have the entire sync and back porch regions, which was around 300 cycles on the 65Mhz pixel clock, to fill our buffer.

Our concept was to have the read requests sent out on the PLB on the 100Mhz PLB clock and continue until the correct number of transactions was completed and wait until the horizontal counter became 0 again indicating the beginning of the hsync sync region. The tft side would be independent and would simply start reading from address 0 of the BRAM when the hsync hit the active region and continue reading until the counter leaves the active region. This would depend on the reads from the PLB being faster than the tft could read from the BRAM.

After we tested as much as we could using modelSim by coding up Xilinx primitives and using a replacement BRAM, we then implemented it on the actual FPGA. After all the counters and obvious problems were fixed we saw an image that looked like the background we intended to display. However the resolution was only half of the actual resolution we wanted to display at. The problem was that our assumption that we could read from the PLB and write to the BRAM faster than the reads from the BRAM on the tft side was wrong.

The original vga framebuffer had 80 transactions of 8 words each meaning the 64 bit PLB bus would give data 4 times per transaction for a 640 pixel line. We simply increased the number of transactions to 128 to get 1024 pixels of information. For one transaction you would need to raise a request line on the PLB until an ack is received and then a certain number of cycles would pass between the ack and when the data and data ack were raised. This wasted many cycles between requests and between the request ack and the data acks. When we changed to a read request terminated by the master we only used 512 cycles on the PLB and the entire background was displayed as we expected it to be.

The next part was to actually create the overlay. We would insert each pixel in real time by using the vsync counter and hsync counters to determine where on the page we were. The block sprite and numbers were compressed into 13 and 5 different RGB values respectively and then chopped into 18 bit colors. Counters were used for blocks and numbers and the correct color corresponding to the counters would be sent to the tft pins. Getting the counters to be correct required chipscope and a lot of testing. Each of the regions of level, next piece, score and the board are defined by their bounding hsyc and vsync counters and each number or block is defined by counters counting the width of each. Counters for the y coordinate of the sprites were updated when the whole line finished updating. Using this method the high resolution screen was displayed (example shown in below in figure 5).

Figure 5: Blank board background

**Software Logic**

Originally the software handled repainting of the board and keyboard presses. After moving the keyboard handler to hardware, the software was used for repainting the board, networking, and audio. There was a performance problem with trying to accomplish all three tasks without near perfect time slicing as certain calls like memmove would take up a long contiguous block of time. This caused the display and audio to be choppy, resulting in a sluggish user interface. The display required 60 updates per second and audio was required to fill the AC97 buffer continuously as sound was played.

A solution to time slicing was using the Xilinx microkernel which allowed all functionalities to be threaded separately. Audio, networking, and refreshing were each placed into its own thread. The scheduling algorithm used was round robin with each thread yielding to the next after the completion of an iteration.

After the vga controller was completed in hardware, the software was alleviated of refreshing the screen. The final version of the software had only four tasks.

1) Initialization of dram with data from the flash card (held images and audio)
2) Audio through controlling the AC97 Codec
3) Networking with other boards
4) Translation of the binary score and level to decimal digits for the vga controller

Our software allows us to read from the Compact Flash and transfer the data into the DRAM. The Tetris background images and in-game audio will be initially stored on the card. We are using the SysACE library to perform read functions that will move the data to a buffer. This buffer will then be transferred to memory using the standard Xilinx IO library.

Audio

The audio output is implemented in software. A software register contains a command indicating which audio file should be played. There are several structures in the software storing data for the audio effects. A WAV struct contains the wave file's memory address, length, position indicator, and play flag. The position variable indicates what packet of the wave file to read. The play flag is set to zero when the sound should not be played and set to one when the sound should be played. Initially, the position and play variables are set to zero. There is also an AUDIO struct that contains all the WAV structs for each audio effect.

The audio function first reads the software register. Then, the flag of the audio effect that corresponds with the command will be set to 1. For every audio effect flagged to play, a packet is read from the DRAM. These packets will be added together and resulting summation is then sent to the AC97 FIFO to be played with the audio codec. The position variable for each audio effect will increase every time a packet is read from the DRAM and then sent to the FIFO. Once the position variable reaches the end of the audio file's length, the position and the flag are reset to zero. This indicates that the sound effect is finished playing. Packets will continue to be sent to the FIFO buffer until it is full. Once that happens, the PPC is free to perform other functions. When we return to the audio function, it will continue to read packets from where we stopped because of the position variables.

When the game is started, the software register is set to indicate that the theme music should be played. When an action occurs, such as a tetris, the software register is indicated to play that action's sound file. Packets from the theme sound file the action's sound file are then added together and sent to the AC97 FIFO. This sound mixing continues until the action sound is finished playing. As more actions occur, those sound files also can be added in. If the theme music position variable reaches the end of the audio file, the position is reset to zero, however the theme music is still flagged to play, thus repeating the background music.

Networking

The network implementation uses polled, send, and receive commands from the Xilinx Emac library to keep the PPC in non-blocking mode. That is, when there are no packets to send or receive, the PPC is free to perform other functions such as audio. The sharing of time between networking and other PPC functions is accomplished through threading.

Our building blocks for networking include a function that checks if there is a frame to receive. If so, it is pulled and stored into a character array. We also have

functions that will generate packets with the necessary information and send them over the network. Our packets generally denote their purpose in a 2-byte header.

Prior to the start of the game, boards may be connected to the hub. Once a master board is set, all other boards will automatically announce themselves to the master with a packet that includes their MAC address. The master keeps a table of all boards so that it can send either updated information or a start game signal. Whenever a new board joins the network, it is the master's responsibility to broadcast the updated table to all other boards. Once a user on the master board presses the start game button, the start signal is sent to all boards. At this point, new boards that are connected will not be eligible to join the game. They must either play on their own or wait until the current multiplayer game ends.

During the game, the networking implementation polls a register on the board to check for line eliminations to send. If it detects that there is a new line elimination, it broadcasts this through the network and players on all other boards will receive lines. In addition, if the software detects that both users on a board are dead, it announces this to the network. Since each board has a table of all other boards on the network, it is easy to tell if there are no surviving network players remaining. Under such a condition, the last board determines itself to be the winner and ends the game.

Our networking implementation is flexible in that it allows the players to engage in different types of games (single player, 2-player on one board, teams games over the network) at any time. When a network game ends, all boards are free to leave the network if they choose and start a single board game. In addition new boards may join once the current network game ends.

**Methodology**

In designing Tetris on FPGA, we first started by obtaining starter code for the Tetris game in Java. This code was incomplete and required the addition of functions for rotate, drop, and line elimination. After we got the Java version up and running, we were able to get a better idea of what was required to build the game in hardware. This included a general list of states that we would need for the main FSM.

Since the game play FSM is essential, we worked on the Verilog for this part first. Our FSM came out to be 32 states in the end after several revisions. First, we coded a single-player Tetris FSM. Once we ascertained that this was working, we moved on to multiplayer so that two people could play on one board.

Once our single player mode was functional, we moved on to interfacing the peripherals. Initially, we were considering using the PPC to control the VGA frame buffer. This decision was due to the ease of debugging our game code logic while it was in a synthesized state, which is very different compared to debugging in Modelsim. While this worked adequately, we found that game play was considerably slower as a result of the bus latencies across PLB and OPB as well as the necessity of sharing

processor time among several features. We were also afraid of running into issues by having VGA, audio, and networking all on one PPC.

Our options were narrowed down to using both PPC0 and PPC1 to split up our code or to move more of our code into hardware. We chose the latter, deciding to create our own VGA controller which would support a much higher resolution and remove the burden of triple buffering in software through memory moves. From discussing the first option with other lab groups, we found there was very little documentation about getting multiple PPC's to connect to the busses correctly.

While the hardware VGA code was being developed, we worked on implementing networking and audio. Since they shared one PPC, we first decided to use a simple form of time-splicing to combine the code for both features. This didn't work too well for some reason, presumably poor handoff timing which probably caused some peripherals to take up too much CPU time. The AC97 buffer was not filled at a fast enough speed nor was the game display refreshing at 60 frames a second. We scrapped this idea and used the Xilinx microkernel which supported multiple threads. This resulted in perfect synchronization of all the required components resulting in a smooth user interface.

Once the hardware VGA was complete, we integrated it with the rest of the project. When we got everything working, we focused on adding some finishing touches. We mapped out some higher quality Tetris blocks as well as numbers so that we could display the score.

During the course of working on this project, we learned a lot about integrating components in an efficient manner for the Xilinx board. Patrick was responsible for the game code, Mai-Anh was responsible for the audio code, David was responsible for the video code, and Edward was responsible for the networking code. When it comes down to integrating all the components together, it was important that at least one person in the group had a solid idea on how all the pieces should and will fit together. Before work is distributed, it is important that each person knew the interface and synchronization signals of their "black box" component.

One of the main lessons from this class was how to keep the project well coordinated despite working separately on different modules. One of our primary wishes if we were to redo this project would be better documentation for some of the Xilinx board features. The pre-project labs were from Xilinx demos and were poorly documented.

Our current status is that we have a fully functional Tetris implementation. This includes sending lines to opponents, mixing audio effects, networking multiple players, viewing next piece and score. Following the public demo, we spent time ironing out the problems that had lingered. This included the add lines feature of networking and correcting the flipped display of the Tetronimo in the next piece area. Scoring is now based on both level and number of clear lines.

In the future, it may be interesting to move audio and networking to hardware and add some features such as changing backgrounds and multiple networked games going on over one hub. We could also allow players to enter high scores and have them stored on the CompactFlash card. Since Tetris is a game with relatively few rules, there are few non-cosmetic features that could be added unless we wanted to make the game play different. Overall, we believe that our design simulates fairly well the feel and look of a production quality Tetris game.

Project website: http://www.andrew.cmu.edu/user/davidfu/

Individual Page: Patrick Chiu

What I Did

For design review one, I completed a java version of Tetris originating from a Stanford undergraduate programming assignment. This serves the purpose of familiarizing myself with the internals of Tetris game logic.

Together with David, we architected and designed the game core logic. Initially, David and I both designed a separate game core logic, these were the first two versions of the design. We decided to use the better performing one after synthesis. In the end a more compact and faster design was needed. I architected and redesigned this third attempt which turned out to be the starting point of our final design. From then till the end of the project, I added to this design all the additional features such as one versus one mode, networking mode/support, add lines, audio support, score/level, and integration with the custom VGA controller.

At this point, I designed the system level architecture, where each component would go, how it was connected, and the signals communicated. As more and more of the design moved from software to hardware, I had to revise and re-plan the system level architecture.

Prior to writing our own VGA controller and integrating a custom PS/2 keyboard controller, I wrote the software for driving the Xilinx hardware and hardware/software integrate. I ported David's initial VGA controller to run on Modelsim and also wrote a test bench that acts as a dummy PLB bus to accelerate his debugging. I moved the keyboard interface to hardware by integrating an opencores PS/2 keyboard handler into our design and interfaced it with the game core logic.

As the system level architect and the owner of the game core logic, I was tasked with creating the specifications for the networking and audio parts of the software. These were then given to Edward and Mai-anh for implementation. They created standalone projects for these individual software parts. On completion, I integrated them with the existing EDK project. The software integration and time slicing didn't go as smooth as I thought which forced me to convert the existing software side of the project into a threaded model. I solved the problem by converting the software design into a multi-threaded model that runs on the Xilinx micro-kernel.

Specifically I wrote, tested, and debugged all of user_logic.v, minus the keyboard interface (from opencores) and random number generator (birdcomputer.ca). I handled all manipulation of the EDK project and system integration. I also wrote all the software (minus networking and audio).

Provided lots of support and debugging help to other team members.

Time Spent

4-6 hours a day before Thanksgiving.
8-10 hours a day after Thanksgiving.
More on weekends

Individual Page: Mai-Anh Duong

My responsibility was to write the code for the audio mixing and outputting. My first task was to extract the data from the flash drive and put it into the DRAM. This took a few weeks to figure out how to extract the data properly. This is because once I had written the code to read the from the flash and store to memory, I needed to play back the audio to confirm if I had read the wave file correctly. Initially I attempted to do audio in hardware. After several weeks of running into problems with sending and receiving all the audio packets, I resorted to implementing the audio in C.

It turned out that I had not extracted the data properly because I had some confusion about the big endian and little endian encoding in the WAV file. Once I had fixed that I observed that the audio file took a very long time to load. So to speed up the loading process, I edited the reading functions provided in the audio lab to read the file buffer 4 bytes at a time instead of 2 bytes.

Since audio was now going to be done in software, I wrote the audio function to behave as independently as possible, so that it could be added into the project as easily as possible. To do this, I decided that the audio would constantly read a register indicating what audio should be played. This worked out well since Patrick had decided to integrate the audio function with his code that way too. Also, I created the structs to contain each sound effect's personal information. This included where in the wave file the audio was currently playing. This enabled the audio function to stop sending packets to the AC97 FIFO, and then, upon returning to the audio function, resume playing from the last played packet.

While also working on the audio, I was responsible for acquiring all the images for the graphics. Since no one had any strong opinion on the theme of the background, I decided to continue the theme of our team name by having all the backgrounds involve penguins. David protested and requested a block background but I was already intent on penguins. So, with the help of surfing online for penguin images and using Photoshop, I created all the background images for the game. However, the attribute of the background changing to indicate a level change was not implemented in time for the public demo. I also extracted the RGB values for all the bits of the block sprites and number sprites to be used in the video implementation. In retrospect, I should have written a program to do that for me instead of wasting many hours.

Overall, I was satisfied with the class. I thought the project was a fun idea and made for an entertaining public demo. However, I feel as though the class's goals and focuses were to code a game, not an FPGA. Also, despite all the problems with EDK and Xilinx, I do not think they should be changed. Students who have taken this semester's course will be familiar with the program and the board, and therefore will be able to help out more than the TA's could this semester.

Individual Page: David Fu

I wrote the initial game code in hardware using registers, tested it and came up with the initial hardware design. I helped design the data structures for the pieces and game code. I tried to optimize the game code but the design used too many registers and therefore too many of the slices of the board. This took around three weeks.

I wrote the vga framebuffer to support 1024x768 and removed the dcr interface since it was no longer needed. The vga framebuffer consisted of a hsync & vsync module, plb interface and cross clock domain logic. I redid the hsync and vsync counters, interfaced with the plb for a read burst until the master terminated and used glue logic and a bram for signals that cross clock domains. I then took the background and sprite information that was given to me and converted them into a ROM type structure to overlay on top of the background as the tft clock. I setup all the counters needed to output the level, score and blocks and discussed the interface with Patrick to get all the necessary signals from the game code to the framebuffer. I created the framebuffer.v, xvga.v, overlay.v, sprites.v and plb_if.v. The tft_if.v file was adapted from the original frame buffer code. This took approximately four weeks.

Individual Page: Edward Liu

My primary responsibility for this project was implementing networking. We discussed several ways to do this and decided to use software running off of PPC0 and the Xilinx Emac interface. The Emac library contains primitive send and receive functions which allow you to make and format your own packets. Now that I had a basic idea of how networking would be implemented, the remainder of the brainstorming prior to coding came down to what kind of features we wanted to support.

The consensus was that we wanted to allow teams of two to play against each other. In addition, as many teams as the hub can fit should be allowed. It was decided that the best way to do this without significant hardware modifications was to make each board a team. I coded my software to meet these specifications accordingly. The router/hub that I received would allow for up to eight players total in teams of two.

My goal was to make networking as transparent to the game board as possible. Therefore, I made sure to keep track of as much information in my software as I could. Data such as the MAC addresses of all boards currently on the network were kept in a table and updated with each new board. Also, the number of dead boards was stored in real time. This allowed many features of the game, such as add line and end game, to be decided by the networking implementation rather than the hardware.

To start things off, I looked at a networking example provided on the Xilinx website which uses the PollSend and PollRecv functions in loopback mode. Once I was familiar with how these were used, I tried coding up a new implementation to send a packet with given information to another Xilinx board. As I connected more boards, I realized that there was no way to keep track of how many had joined the network and what their individual MAC addresses were. This is where I got the idea of having a master board with a known starting address. The known address would allow any board joining the network to announce itself.

I was also faced with the issue of what to do after all boards were connected and the game started. I decided that the best way to handle this would be to make both master and slave boards leave their respective blocks of code and merge into the same code so that they would no longer be distinguished. This guaranteed that all players would receive the same functionality during the game and simplified the debugging process significantly. Once the game ended, I allowed my main networking function to end so that all previous game data would be flushed. This also allows new boards to announce themselves and join in for the next game.

Since the public demo on Thursday, I fixed the problem we were having with the add line feature. It now works correctly and all boards/players on the network receive the proper number of lines. The problem that I was experiencing involved switching between 32 and 8 bit pointers (since the data that I needed to transfer came in 32-bit format). Once the pointer arithmetic issues were resolved, the networking module was fully functional in starting, restarting, and implementing all intended features.