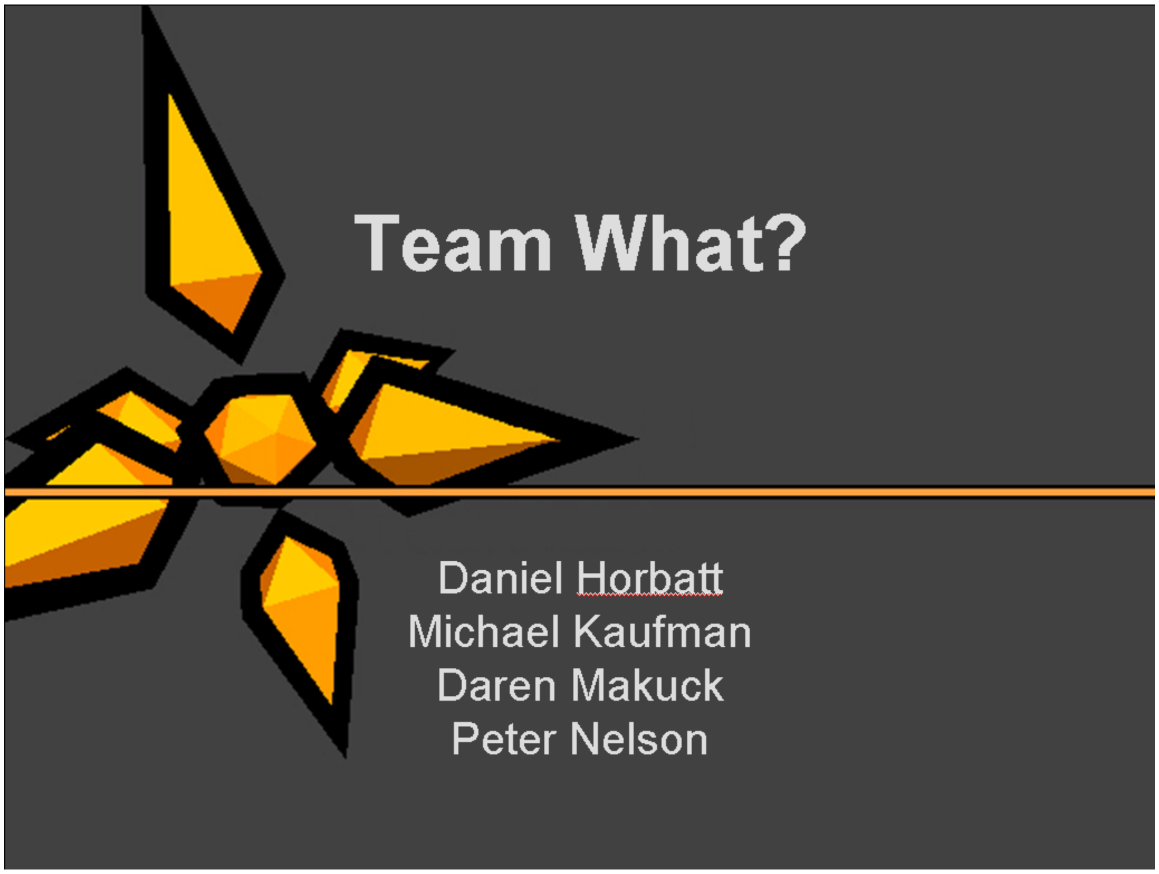


Team What?

Daniel Horbatt
Michael Kaufman
Daren Makuck
Peter Nelson

I. What We Built and How We Built It

A. Game Description

This game is an overhead action-strategy game in which you, the player, navigate a ship that is a smart bomb around the screen. At the press of a button, the ship's self destruct mechanism is triggered and it explodes on the screen. Randomly flying around the screen are several enemies that when caught in the explosion from your ship, also explode. The player can link these explosions if the other enemies fly into an explosion on the screen. By 'chaining' these explosions, it is possible to rack up high scores and get power-ups that will speed up the generation of enemies, give the player a time extension, or give more points. Once a player has obtained enough points they will receive an 'extend' and have one life added to their stock. The amount of points required for an extend then increases by a set amount based on the selected game mode.

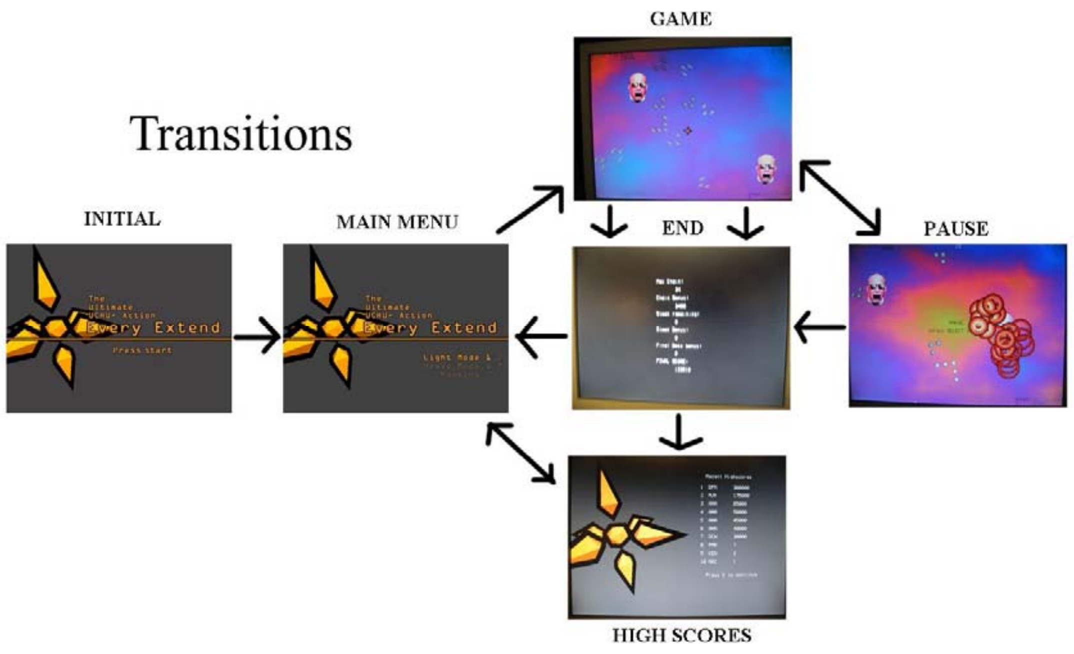
There are three type of 'normal' enemies in this game. There is a red enemy that will appear on the screen by itself, and when killed will drop a 'quicken' powerup. The 'quicken' item will increase the number of groups of enemies on the screen by one, up to a maximum of eight quickens, giving a maximum of 70 total enemies on the screen at once (not including bosses). The green enemy flies at the front of the formation of any other group of enemies (except for when the final boss is on-screen), which includes groups of three and five enemies. Once the green enemy is destroyed it will drop a normal powerup which gives an increasing number of points, starting at 800. If more than one of these powerups is obtained before either player dies, the amount of points received increases by 800 over the previous value, thus giving 800, 1600, 2400, etc., points. The final 'normal' enemy is grey-colored and does not drop any powerups. This enemy is also the most common one found in the game and fills out the rest of the V-formation for groups of three and five that are headed by the green enemy. Each 'group' of enemies has a different chance of appearing. The red type enemy 'group' has a 1-in-16 chance of appearing, the group of three enemies has a 1-in-4 chance, and the group of five enemies has a 11-in-16 chance of appearing.

In this game there is only one ‘level’, however the length of the game is determined by a countdown timer and the number of lives the player has. The timer can be refreshed by a finite amount during the game if the player gets certain power-ups, which appear after defeating either of the two ‘minibosses’, and after it has run down far enough, a final boss is triggered which the player must defeat to gain an even bigger score. The minibosses and the final boss can only be harmed by chaining explosions from other enemies onto them. Each type of boss has a certain number of hit points (2 for the minibosses and 4 for the final boss) which are decremented each time a set of explosions is chained onto it. When the final boss appears, no more powerup-giving enemies will be created until the boss is destroyed, thus making the focus of these final moments of the game on the boss itself rather than distracting the players with trying to get more points and powerups.

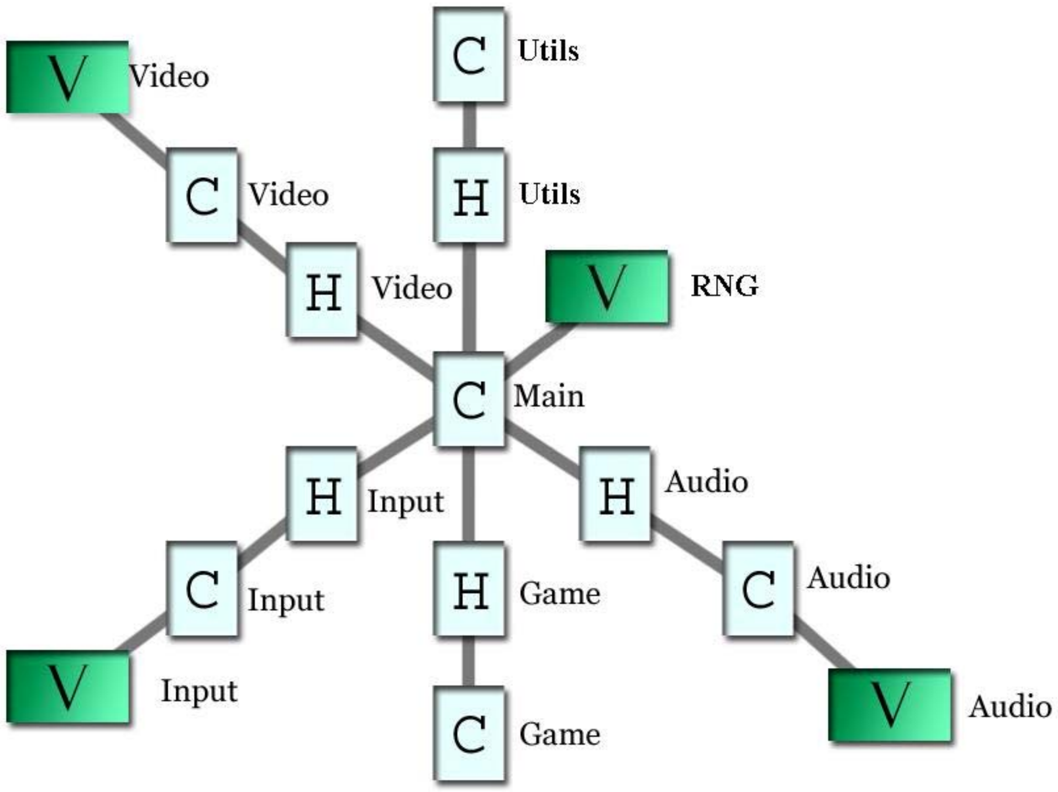
There are four different modes of gameplay, as well as a rankings option. First, the player may choose ‘Light’ or ‘Heavy’ modes. In ‘Light’ mode, the required score for the first extend is 1000 and each extend after that requires 2000 more points than the previous extend. In ‘Heavy’ mode, the required score for the first extend is 4000 and each extend after that requires 8000 more points than the previous extend. The minimum speed in heavy mode for enemies is one more than that of light mode, however both modes still have the same maximum speed (seven, in either the x or y direction). The final difference between heavy and light modes is that in heavy mode the final boss will ‘replicate’ once he drops to 1 hit point. At this point, the boss shakes for a few seconds and then splits into two versions of himself. After the two heads have split sufficiently they will then continue flying about the screen in a random direction. What makes heavy mode even MORE difficult (as if having to kill one of them wasn’t hard enough with the time restrictions) is that the replicated head is completely imaginary and does not take damage (although it does not harm the players, either). Thus, the players must determine which head is the real one before their stock and time runs out.

Aside from ‘Heavy’ and ‘Light’ modes, the player may choose to play either mode with a second player cooperatively.

The multiplayer aspect of the game allows two players to play together and try to obtain a high score. In this mode the score is shared and there is no danger of accidentally killing the other player. The stock in both two-player modes is doubled, giving a total initial stock of 24, as opposed to only 12. While the score at the end of two-player mode is not saved for future use, it is still quite an addicting game to play. One of the best times that we, the creators, had was when two of us obtained a chain of 93 during a multiplayer game.


GAME
Transitions
INITIAL
The Ultimate UCHU- Action Every Extend
Press start
MAIN MENU
The Ultimate UCHU- Action Every Extend
Light Mode &
END
PAUSE
HIGH SCORES

B. Hardware/Software Partition



Our game has been first partitioned down to various libraries which are then further broken down into either software or hardware partitions.

Utils: This has our compact flash memory reading functions, which are done in software.

RNG: This contains our random number generator, which is implemented in hardware.

Input: This library contains all controller polling and rumble functionality, and is implemented in hardware.

Video: This library contains all of our sprite imaging and text placement on the screen. It is implemented in software. It also contains our dynamic, fractal-based background, which is implemented in hardware.

Audio: This library contains all interfacing with our 16 channel mixer. It is done in hardware.

Game: This library contains all of our subroutines required to update the gamestate and the entities that currently exist in the game, as well as the collision detection for objects. It is implemented in software.

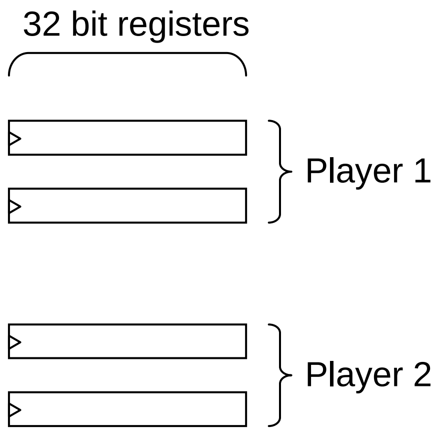
C. Hardware Description

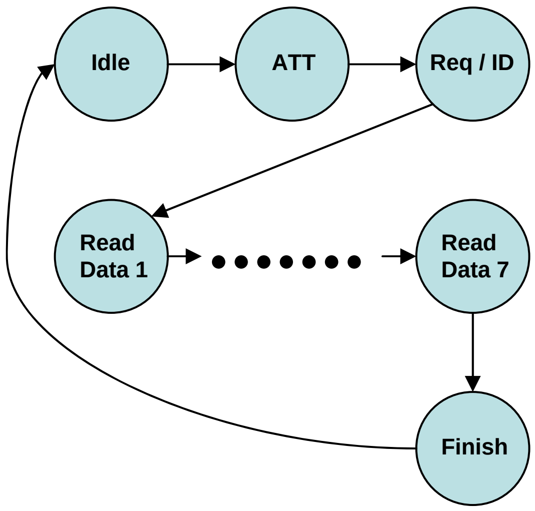
RNG

The random number generator, which generates a random 32-bit value every cycle, is used to establish randomness in obstacle starting point, speed, color, and also the background. It's the XOR of a Linear Shift Feedback Register and a Cellular Automata Shift Register.

Input

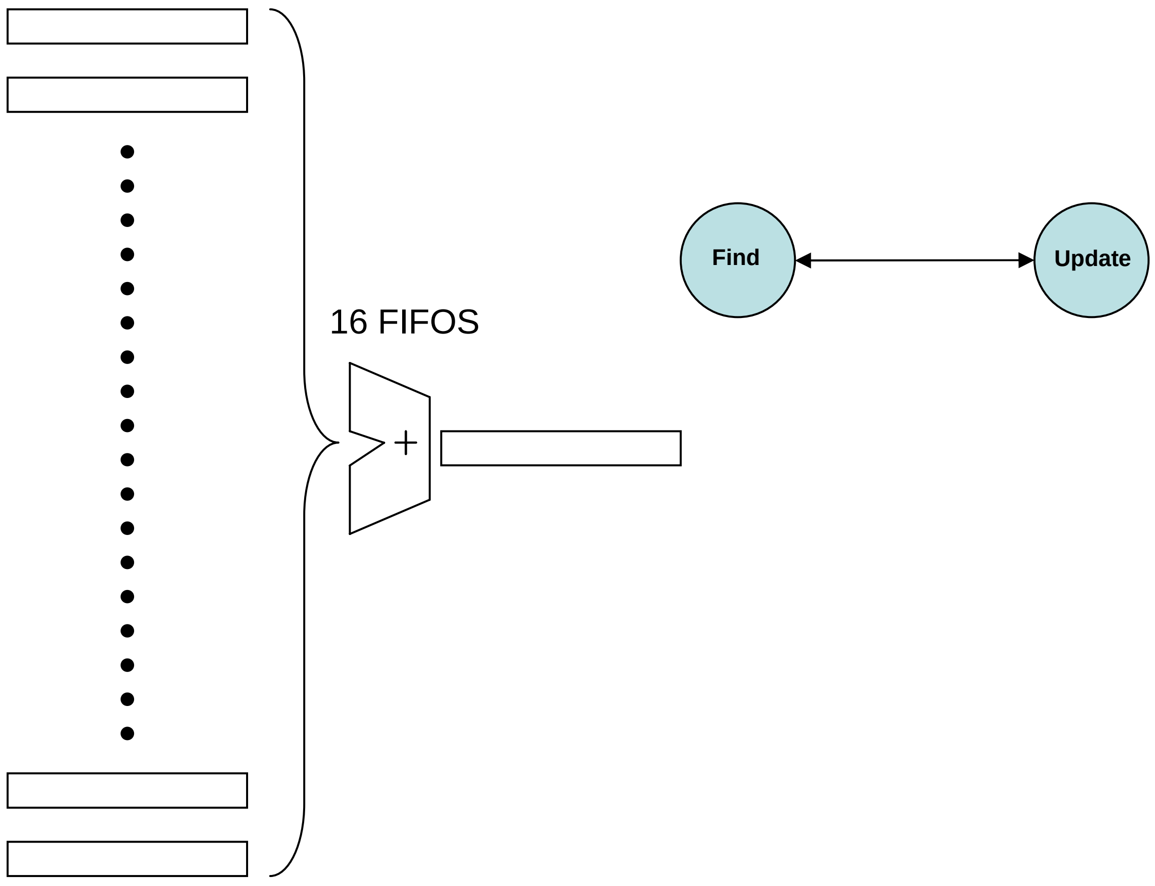
The input for the controller contains four different 32 bit registers, two for all the buttons and analog sticks of each controller. The module slows the clock speed down to be compatible with the controller and uses an eleven state FSM, of which most states are simply reading data. This layout is roughly shown in the diagram below:

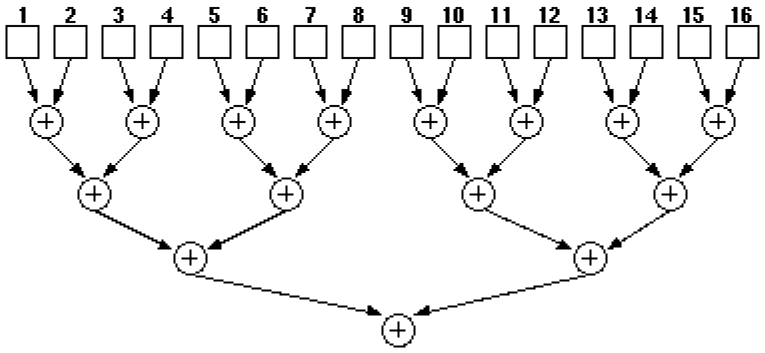




Audio

The audio is implemented using sixteen FIFOs to contain a maximum of sixteen different channels to be mixed together by simply adding the data, while taking into account any overflow or underflow. The module flips endianness by rearranging bits so that it does not need to be done (slowly) when the audio file is read. There is a simple 2-state FSM that checks if any channel needs data read for it and then reads data of that channel. Originally data was writing across the PLB to the AC97 codec, but Xilinx interface code would lose about every 1700 writes causing static. This was fixed by modifying the AC97 codec to expose the internal FIFO and having the mixer module write directly into the FIFO.



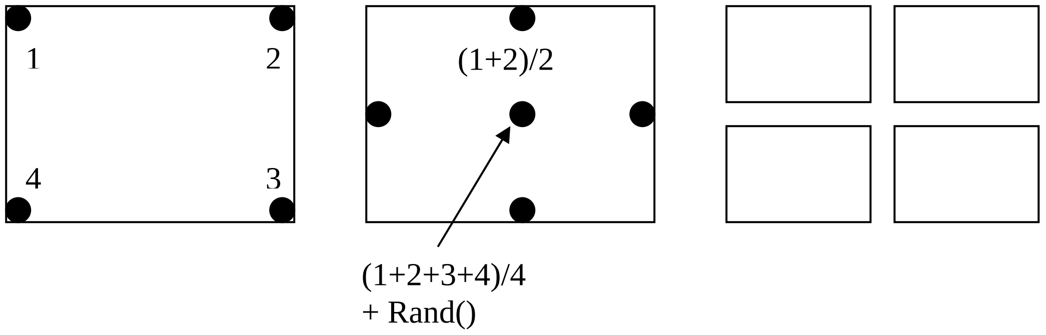


Or tree of adders to reduce critical path from 16 adders to 4

Video

All of the game-related video is handled in software. In hardware we have a module that was meant to blank the framebuffer since it was taking a significant time in software. Instead of just blanking the background we created animated fractal plasma.

The basic algorithm is you start off with the 4 corner points at random values and then draw the midpoints of the edges as the average of the points on that edge. The center point is then the average of the 4 corners plus some random displacement. The algorithm is then called recursively on the 4 quarters of the image, adding a smaller displacement at every level to produce a smooth image.



D. Software Description

Utils

The memory reading functions simply call sysace_fread to read an unsigned short or an unsigned int in big- or little-endian order. We transfer sound and image data from the compact flash to memory, parsing out the headers first. We have structs for each of these types of files which are stored in the BRAM that contain the information about each data file. From the headers we can figure out the width and height of the images that we read in and store that information and the address of the image in the BRAMS, whereas the actual image data is stored in memory. For audio, we store the file length and address of the file in a struct and send it to the BRAM and store the actual audio data in memory.

We have to initialize the audio and video settings so that they can be used by the rest of the program and hardware.

Video

For video, we have three buffers set up and swap them as necessary. Each buffer is 2 MB in size, and we have one register that keeps track of the address of the buffer that the game should be using to display images. Since all of our images are already stored in memory, we just need to load the data into the appropriate frame buffer. Since the frame buffer actually contains enough data to display a resolution of 1024 x 768, each pixel needing four bytes of data, and the resolution that we actually display is only 640 x 480, we need to make sure that we only store data in the appropriate part of the buffer. We only write three out of the four bytes for each pixel, and discard the transparency byte from our stored images when we write to the buffer. We still, however, write four bytes (we write 32 bits at a time to the FIFO) as we treat the transparency byte as a zero. We also flip the endianness of the image data so that it is displayed correctly.

The text on the screen is written by a function which determines which pixels need to be written to based on a given address and the ASCII code given. This text function is used to write information about the player's score, the time left in the game, and quicken values.

We swap frame buffers every frame, which also correlates to anytime something changes. Therefore, if the player moves, objects move, the player explodes, etc, we will need to swap the frame buffer. Each frame in our game, the state of each entity (enemies, players, powerups, score, etc) is updated accordingly, and the frame buffers are swapped.

Game

Our game code contains the procedures that we need to make the game run. This includes making calls to load the images / audio into memory, loops for the game menu(s), gameplay, displaying / updating the player, explosions, and the flying objects, power-ups, etc. Since almost everything else is dealt with in hardware / other modules, this code is mainly just calls to those functions.

Collision detection is handled in software. Our original thought was that the software would not be able to handle the extra calculations, however after we implemented it we found that it was a lot easier and quicker than we had imagined. The module is given x-y coordinates for the enemies, explosions, and the ship/bomb. Using this information each enemy is compared to every explosion and the ship/bomb to see if their center points are close enough to cause a collision. If a collision is found, the enemy is changed into an explosion and updated appropriately.

In-game entities are all type-based. What this means is that each object, whether it be enemy, player, or powerup, has a corresponding type. There is also an all-encompassing “empty” type. While there is a separate array for enemies, bosses, players, and powerups, all entities can become the “empty” type. Almost all of these can also become an “explosion” type. When using the collision detection, objects of type “explosion” are compared with the enemies and bosses that are of their corresponding enemy and boss types to determine whether or not they should explode / lose hit points. Player types are also compared with enemy and boss types to determine whether or not the player needs to be changed to a penalty type (which is basically the same as an explosion type, except the penalty type doesn’t get compared for collision detection and thus cannot harm enemies). Since each entity has a type, it is easy to determine which sprite should be displayed for each entity given their type and a sprite number, which just tells which “frame” of the animation needs to be loaded for a given entity.

Enemy generation is based on “groups”. A cap is set on the number of total enemies allowed on the screen, and an array is created of said size. There is also a “soft cap” created on the number of enemies on-screen that is based on an initial cap (30 enemies) added to the number of quicken items received * 5. Since the enemies are “group” based, each enemy is updated with the other enemies in the same group. Basically, the array that contains the enemies is checked by an index that is incremented by 5, and any time every single enemy in the group is of the empty type a new “group” is created. The type of group that is created is based on a random number generator (0 is a single red enemy, 1-4 is a group of 5 enemies, and 5-15), and the location and direction that a given group will appear from and head towards is also randomly generated. The maximum x and y speeds are capped at 7, which means that in any given frame no enemy will move more than 7 pixels in either direction. If the type of group is not a single red enemy, the other enemies are placed based on the the initial location and direction / speed that the leading enemy is placed with. If the final boss is active, only groups of 3 and 5 are created, and the head object in each group that gets created is set to grey instead of green. We treated the front object’s initial speed and direction as a vector, and used this along with a pre-calculated scale factor (which we store in a lookup table) to place the other enemies behind the first in increments of 32 pixels behind and to either side.

Powerups are created once a miniboss, red enemy, or green enemy is destroyed. The speed of a powerup is pre-set and the direction is changed to the opposite direction with which the enemy that was destroyed was heading. If the player collides with a powerup, the powerup is changed to the empty type and the corresponding bonus is applied. The maximum number of powerups which can be on the screen at a given time is 1/5th that of the maximum number of enemies. If we reach this cap, then the first powerup in the array will be overwritten.

If a player collides with an enemy, he / she is then changed into a penalty type and reset after the corresponding explosion has expired. Once the player is reset (assuming there are more lives remaining), he / she then has 20 frames worth of invincibility with which to move around. The player cannot be harmed by enemies, however he / she cannot blow up their ship, either. If a player runs out of stock, however there are still enemy explosions on the screen, the game waits until the explosions are gone and will revive the player if the player receives an extend due to these explosions. Also, if there is no remaining stock in two-player mode but one of the players is still alive, the game will wait until the explosions from that character have ended to determine whether or not the game is over. If the timer runs out at any time, regardless of whether or not the player is still alive or there are explosions on the screen, the game will end.

Minibosses are created at 30 and 60 seconds into the game. If one is “hit” once, the direction the miniboss is spinning will reverse (which is just a simple decrement of the sprite number as opposed to incrementing it). If it is hit again, a large explosion will result which can blow up enemies, and a time powerup is created that heads towards the middle of the screen. The final boss consists of 4 “stages” (basically one for each hit point). The final boss initially enters from the center of a random side of the screen, and then moves diagonally up, down, left, or right. The final boss will bounce around the screen until killed or time runs out, instead of disappearing after it goes off-screen like normal enemies. Each time the boss is hit, his frame number changes and it is incremented in a different range. Both the first and last “stages” of the boss are just one sprite, and so the frame number isn’t incremented at all. During heavy mode, once the player gets to the final stage of the boss a “shake timer” starts and the boss is placed on the screen and random numbers between 0 and 15 are stored in shake_x and shake_y variables each frame to give the boss the appearance of shaking. After a set amount of time, a “fake” boss is created and the two copies of the boss are sent in directions perpendicular to each other for a set amount of time, and then their directions are set diagonally again. The only updating done for the “fake” boss involves changing his position, since he can neither blow up the player nor be blown up. Once the player destroys the real final boss (in any mode), the boss shakes again and then explodes. A “you win” message is then displayed for a set period of time, and the game resumes normal play until either the time runs out or the player runs out of lives.

After each first player game, the final score is compared against the high scores list and updated accordingly. This list is reset every time the game is turned on, so scores will only persist until the power is turned off or the board is reset.

II. Schedule

Initial to Checkpoint 1:

From the time the class began until the first checkpoint, a lot of theoretical work was done concerning how the game would be made, what features we would implement, and a timeline for our project. We also spent time in lab getting the AC97 audio codec to work so that we could load and play audio files from the compact flash. After we got the audio working, we had completed one of our goals for the first checkpoint. Unfortunately, we were still waiting on a PS2 controller to test out our input drivers, so we were slightly behind schedule.

Checkpoint 1 to Checkpoint 2:

After the first checkpoint we received PS2 controllers so that we could add input to our project. Drivers for the PS2 controllers had already been written by one of our members for a previous class, so we spent the rest of the day hooking up the drivers and getting them to work with our board. After we got our input working, we started work on the video aspect of our project. We made it so that we could display any image on screen at a given location, and then wrote code to allow us to move a "player" image around the screen based on input from the PS2 controller. We created a double buffer for our game so that each frame would be drawn seamlessly and without the choppiness of single buffering.

We started work on our audio mixing in hardware and set up the 16 FIFOs that we would be using to mix audio samples together. We got pretty much all of it working except there was one error with the code: when doing single writes across the bus, writes weren't happening even though they were getting acknowledged, and so we ended up with sounds that had random noise in them. This error wasn't fixed until after checkpoint 2, however every other part of the sound mixing worked so it was pretty much complete.

At this point, we had completed all of our goals for checkpoint 2 (as well as the unfinished one from checkpoint 1). We decided to do a little more work, however, and add in collision detection in software (although we were still planning on doing it in hardware at the time) so that we could collide our ship with the object which we created. We also created a random number generator in software so that we could place the object randomly.

Checkpoint 2 to Final:

With our input, audio, and video working, all that was left was to write game code so that we could actually play our game. Between checkpoint 2 and our final demo, we wrote our game code to initialize the game state, generate enemies, create bosses at the appropriate time, detect collisions between enemies, players, and powerups, revive players, update score, extend, time and stock information, update the rankings and enter initials, and end the game at the appropriate time.

Besides the game code, we also did work on our fractal-based background and finally fixed the problem with our audio mixing, leaving only a small amount of polishing needed between the final and public demos.

Final to Public Demo:

Between the time that we gave our final demo and public demo (2 days time), we finished our fractal-based background and added code to make the heavy mode final boss harder (aka we added the replication portion of the boss).

III. Individual Contributions and Thoughts

Daniel Horbatt

Initially I worked with Mike on getting some sound working. We first tried getting it done in software, but due to complete lack of an API or instructions on the small nuances of how everything truly worked, this was a much harder task than originally estimated. We attempted to get it to load an entire wave file into memory and then pass that to the AC97 chip, but for some reason the file was not being saved properly. We found that out to be a result of endianness and a few other weird things from the file header, and fixed that up. There was also the problem with caching on this. The caching resulted in lots of fun trying to debug things, as what we were fairly positive was being written to memory from the compact flash was actually just being cached somewhere, and when then AC97 chip read it in, it didn't get what we wanted it to. After finally getting a simple hand generated square wave to play, we managed to figure out all the bugs and get the audio working in software. Due to the large processing time required and the complexity of software mixing of multiple channels, we decided it would be best to implement in hardware, so we passed it on to Peter, our resident hardware expert. Overall this took around a week to a week and a half of nights and weekends spent in the lab.

From this point, we had laid out a full Gannt chart detailing the 4 tracks of the final project that we had anticipated. They were the audio, video, input and game tracks. Each of them was fairly independent of each other until halfway through the game track, at which case the video, audio and input tracks fed in at various stages. This was the first real stage that we were able to plan the splitting of the work up. Daren and I took on the game design track. He initially laid out all of the pseudo code while we were working on the audio and some of the beginning video stuff, after which I took over. I converted the pseudo code into compilable C code, and begin laying out the structure of the various libraries that we would need for the final version. This allowed for the other tracks to have an idea of what kind of API they would have to conform to, as well as greatly speed up integration of the various side tracks. This took a couple of weeks to accomplish of nights and spare time throughout the day, as I had complicated traveling weekends. During this period, I also spent a night with Mike figuring out the basic collision detection system that we planned to use. It was simple circle detection based on the distances of the two midpoints compared to the combined radii of the circles.

After this point, I decided to start working on the final resources for the game while everyone else finished up their individual tracks and began integration into the final project state. Initially we used just basic sprites that we created in MS paint, with only a couple frames of animation. Wanting to make this look as 3D as the original, I learned POV ray and began modeling all the sprites out in 640x480 pixel 3d renders. After this, I collapsed them to the respective sizes, added the transparency mask and broke them down into their animations. After finishing, I created the 104 sprite images used by our final game. During this, I also made most of the splash screens for the game as was needed. Finally, I spent a few hours trolling over Overclocked remix (a video game soundtrack remixing site) to find the two songs that we used in our final version. This entire process took about 3 weeks of nights and weekends to learn and get the finished product.

From this point on, I mainly just assisted whatever Daren and Mike needed for the finishing of the game code while Peter finished up the fractal background. Then of course, there were the hours of play-testing that we all had to do, but I don't count that as work so much.

My suggestions for the course are to perhaps either locate, or create, some API's for a lot of the features that we needed for the project. Also, if any teams had done this sort of thing before at CMU, perhaps give us access to their code so we could see how they accomplished certain things. As we were probably the first to actually use this hardware though, I guess that would not have been possible. For future generations though, let them see what we did and how. It should save them a ton of time and problems. Other than that, I think the course should have maybe another checkpoint, and encourage people to plan out things a little better before hand when they have just started the first couple of labs. The labs, while interesting, didn't really do all that much to help us out other than to get us familiar with the software we were going to use. Maybe just one or two big labs would be better suited for this.

It seemed like a lot of groups had problems with the sheer size of the projects they decided to undertake. We were very successful in what we did I feel in part because we were very honest with each other and chose a project that we felt would be easy even if we had a lot of other work to do. We did this knowing full well that these things can usually get way out of hand if you are not careful, and in the end we were able to deliver exactly what we had promised. Other than that, I highly enjoyed the course and enthusiastically recommend it to other ECE majors looking at capstones.

Mike Kaufman

In the beginning of the course I spent most of my time with Dan getting the audio in software with the AC97 to work properly. This accounted for at least a week of working for about 6 hours each day. From that we got the initial loading elements of our final game. Reading from the compact flash and storing its information in new structures and the actual data in memory. Originally the endianness was swapped to be played directly by the AC97, but when the hardware audio mixer was implemented that portion was removed as the hardware was doing the swapping.

After the first design review, when we received our PlayStation 2 controller, we spent about 12 hours in the lab that night and basically got the input done. My contribution to that was general debugging, some logistics for the FSM, and playing with a scope and reading the signals from the parallel port add-on to the xilinx board. By the end of the night we realized all the problems we had were because of the pin numbers on the board being in the reverse direction that we though they were.

I was responsible for the original work of getting our vga double buffer and the placing of a sprite and then later an animated sprite on the screen. This took a good deal of time as a lot of debugging needed to be done to figure out the logistics of the framebuffer, how things should be written to it and in what fashion. With the double buffer, I decided to have a function to swap the reading of the buffer and the writing of the buffer, this could have been done in one function, swapping reading and then writing in preparation of the next write, but in looking into the future of implementing a triple buffer and blanking the background or placing a background into an extraneous buffer, this method would be more useful. The first beta that I wrote was basically like a giant colored etch-a-sketch or a zamboni-like game, where the ship was controlled by the input and had boundaries of the screen, but it's previous placing was not erased and so as you moved around the screen a trail was left. If I had to guess, I'd say this took approximately a week and a half of 6 hour days, but everything is hard to judge as we continually went from one thing to another.

The next beta version that we would create was to be closer to the final product for the second design review. I wanted to implement a controllable ship, with transparent pixels, and redraw the background each frame in order to remove the etch-a-sketch effect. Also, I wanted to add an enemy/obstacle for the ship to collide with, which meant implementing collision detection, and finally add an explosion animation when there was a collision. This took all the time between finishing the other version and up to the second design review. Any chance I got, I spent in the lab working on this; multiple hours a day, almost everyday for that time period. Once this was finally done, we had a little extra time, and since the framework for the final version was finished by Dan, we were able to port this code over to a new version and add some functionality.

Once we had the basic framework for the final version of our project, I ported the old code to this new version and got it to a point where it had the same functionality that it did in the prior version, but now it was much more organized and ready for all the game code to be added. I made some addition to the code at this point, adding multiple objects, appearing singularly and randomly at the screen edge and moving at randomly set speeds. This introduce strain on the machine because it did not yet have a random number generator in hardware, and we could test the speed of the collision detection algorithm, which was pretty simplistic check the radius of objects and comparing them like circles. After the checkpoint, I explained the code to Daren, and he took over most of the game code, and I helped him make decisions and coded certain aspects that would have been too difficult to explain as my coding style was probably a little awkward. In the final two weeks leading up to the final demo, I spent almost every waking minute in lab, and pretty much lived there. It was a fun atmosphere, and not to stressful as we didn't run into too many extra problematic difficulties that had us ripping out our hair.

Overall, I found this class to one of the best classes I have taken here so far. I got first hand experience with a higher end FPGA board, got to program it, and the most enjoyable thing, I got to code a video game and all its many aspects and see a polished final product from it. Of course there were plenty of bumps on the way; having limited documentation on the xilinx board, notably the AC97, vga, and buses, the poorly written, or sometimes not written, code provided by xilinx for the hardware, made doing exactly what we wanted a very slow and heavily debugging process, and the software provided by xilinx was also a little frustrating to work. If the hardware code provided by xilinx worked a little better, then the project course could have been a lot less strenuous. After finishing, I realized how difficult it might have been to design a game from the ground up and then implement that. We simply would not have had the time to do that, especially since so many different decisions needed to be made about gameplay as we moved on, and we had a version of the game to look off of continually to aid in our decisions. Without a source like this, the project would have been very complicated and we could have run into unforeseen difficulties.

Daren Makuck

For the first part of the course, I spent most of my time helping debug the hardware code and deciding on the logistics of our game. We all met a few times a week for 4-6 hours and worked on the labs, discussed how we should go about our work, or write code. As of the first checkpoint, we had audio working but not input. After the first checkpoint we received our PS2 controllers so we spent about 10 hours that day trying to get it to work. I helped debug the code and hook up the scope to the pins so that we could check the power to each of the pins.

Between checkpoint 1 and 2, our goal was to have our code draw images on-screen and allow a sprite to be moved around the screen by our PS2 controllers. We again were meeting a few times a week for 4-6 hours. Again, I helped debug the code that we used to drive the video aspect of our game. I also helped write some of the code after we got the loading images and displaying them on-screen aspects working. I also wrote pseudo-code for our game and gave it to Dan, who then wrote the initial basis for our game code.

After checkpoint 2, Mike worked on integrating Dan’s code with the code that we already had working. Once that was done, I took over writing the game code. I ended up writing ~85-90% of the game code from main.c, game.c, and game.h (about 2300 lines of code combined), which contains all the code to run the game, as well as ~40% of video.c (approximately 1500 lines itself), which contains all the code to load/store images, display text on-screen, clear/swap the buffers, and update scores / some game information. In the last two and a half weeks of class, I spent at least 250 hours writing this code, spending 10+ hours in lab most days of the week. There were a couple times where Mike and I swapped places, mainly in places where dealing with the hardware code was involved. During the times that we weren't writing code, Mike and I were discussing how to go about certain aspects of our game. There were quite a few times where we just sat down and drew different parts of our game and tried to figure out what the best approach was. While we were writing code, Dan was spending time making all the sprites that we needed for our animations and game menus, as well After our final demo, I spent another 10-15 hours getting the bosses to work correctly and move about the screen / replicate.

In all, I loved this project. I could not have lucked out any better for a capstone project, as my goal in life is to create games such as this. This is the first time at CMU I have ever had the opportunity to make a game, and so it was quite the learning experience for me. The only thing I would change about the course would be the labs. I don't feel I learned much from them as they were all pretty cut and dry. It would have been nice (albeit it probably would have made the course seem quite a bit harder) if we had to write code to get the labs running. I had assumed before we even started the labs that we would have to write the necessary code to prove that we actually understood what was going on, but instead we just had to copy and paste the necessary items and we were done. Even so, though, this was only a minor flaw. As a whole, the course and the people running it were great, and I couldn't have asked for anything more.

Peter Nelson

My contribution was all of the verilog than ran on the FPGA. I touched a bit of software and others helped me a bit on the verilog, but not very much. The modules (in order created) were the PlayStation input, an audio mixer engine, a random number generator, and the fractal background generator.

The PSX input module was based on the Linux kernel's gamecon driver that rewrote a few years ago and got accepted into Linus's official kernel. I already had the adapter to connect the controllers to the parallel port, so that's what I used. It took me a few days to hack out the verilog and figure out how I/O works on the board to get it connected up.

The next module was an audio engine. Basically a DMA engine that reads in up to 16 channels of data, mixes them, and then writes them out to the AC97 codec. The basic module was written, debugged, and working in about a week. It then took almost a month of fighting with the board trying to figure out why audio was dropping out when the module wrote directly to the AC97 codec instead of having the software read from my module and write to the codec. I went through multiple iterations of having the module write to memory instead of the codec, using ChipScope monitors on the PLB bus and internally in the module, and various other debugging to figure out what was wrong. In the end I concluded that Xilinx's provided IPIF (IP InterFace code, provided code to help a module talk to the bus) does not handle some edge case and responds that a transfer is done when it should be retried. To work around this problem I modified the provided VHDL AC97 codec (learning some VHDL along the way) exposing the output FIFO's pins directly as module inputs. I then wired my mixer module directly into the FIFO bypassing the bus entirely. It is obviously not an optimal solution, but it works.

The next module was a random number generator. I threw it in one night because I was bored, it was simple to do, and it's arguably useful.

Finally we were looking what else to put into hardware. I started to write a collision detection module to offload those calculations (but obviously nowhere near the scope of the PPU). After a day of coding and a basic FSM to do the calculations I realized that we would gain very little by doing collision detection in hardware. In the worst case we have on the order of 10s of explosions colliding with 10s of enemies. Even at 100s of cycles for each collision, you're only spending on the order of 10 kilocycles on a 300megahertz clock.

I threw away the collision detection code and started on a background module the day before the in-class presentation. I had wanted to do something shiny with fractals for a while, so I did it. It was about 1 day of prototyping in java, 1 day of creating the verilog for the recursive FSM, and 1 day of getting it working on the board. This module ended up being so easy to write because I fully simulated while writing it (which was not possible with the mixer since it needed to be full-system) and had already spent so much time on the mixer I knew the bus very well.

Overall the class was extremely good. I've never spent much time with an FPGA so this was a great learning experience. The best part of the course though was everyone working in the lab helping each other out until insane hours of the night. As for what I think could help improve the course:

The labs were basically useless. Each one we were handed finished or almost-finished code, and you simply did steps 1, 2, 3 and it worked. They required next to no work so we barely paid attention to them. Some thoughts I've generally had on what the labs should be:

1) Figure out how all the basic I/O works. Mostly a freebie. Start a new project selecting most (or all) peripherals. Provide individual samples of code (or point at the header files) for reading files, setting up the frame buffer, writing to the frame buffer in memory, writing to the AC97, and reading PS/2 input. There will then be a bit of work to combine all the bits into a working program. All groups used parts.
2) Figure out how modules work. Create two or three modules from scratch. One could be a memory / computation block. One easy thing would be write to a memory-mapped ram, then when a go value is written to a register an FSM kicks in, reads in every value and averages it, producing a "done" register and a value register. Another module should do some I/O. Could be something like make an LED blink on a protoboard and read a switch off the protoboard. A third module could be a simple DMA engine that does reads / writes on the bus. All the groups had some sort of computation module. Most had an I/O module (the 3 game controllers, pacman's sensors). I think only Penguins and us had DMA masters.
3) Get the board to talk to the workstation. I don't know much about this since we didn't do it, but is seems that the 3 teems that used networking had lots of random problems getting it to work.
4) Maybe a Linux lab, though it seems that encouraging groups to use Linux is probably counterproductive. It seems that xilkernel would be far more appropriate (and simpler) if groups need something more than the very basic standalone OS.

This is just what I thought of off the top of my head. The basic thinking is get groups to really understand what is going on with the boards and the workflow ahead of time, instead of incrementally figuring it out through the project.

Another small issue seems to be the timing of the course. The first few weeks being spent on intro labs and the proposal makes sense. The problem is that since the first design review didn't have an in-lab demo, it was really easy to BS our way through it without actually having done anything. By the time we actually started working on the project (for DR2 with the demo) it was already halfway through the semester. Having an in-lab demo for DR1 would force actual work to be done by that point.

Finally there's the TAs. Other than creating the initial labs I personally didn't see them do too much work. I remember that at one point there was a TA assigned per group, but I can't remember who they were or if we ever actually talked to them. I ended up helping a few groups with ChipScope (the way from the lab doesn't really work in XPS), a bit of audio, and lots of general debugging because I was around and could answer their questions.

Anyway, sorry if that was a bit rambling, I still haven't slept much since the demo☺. As I said I really did think that this was a great class, I just think it could be polished up and made even better with a bit of work. Also feel free to e-mail me later in the week when I've gotten more sleep.