XilDoom
Final Report

18545 Fall 2010
Robert Waaser
Shupeng Sun
Soo Hwan Kim

Summary

For our project, we ported the classic 1993 first-person shooter Doom to a Xilinx LX110T Field Programmable Gate Array (FPGA). We chose to use a source port known as Chocolate Doom which uses the Simple DirectMedia Layer (SDL) library to achieve cross-platform compatibility. After building a custom computer system on the FPGA that was capable of running the game, we wrote custom drivers for SDL that allowed Chocolate Doom to run on our hardware. The final result was a SystemAce file that could be used to configure an FPGA to run Doom on boot, complete with keyboard input and audio and video output. Networking was also implemented, but not tested before the public demo.

Our Project Idea

For our capstone we were required to implement a game, a game console, or one of a number of research projects. For the game route, the game had to be complete with audio and video output, some sort of user input, multiple players, and a way to keep score. Furthermore the game could not be of our own creation, as the course was not meant to be a course in game programming but rather a course in systems-level design.

We took a look at capstone projects from the past three years and noticed that projects that ported software games such as Quake and Descent seemed to have a relatively high rate of success. Typically these projects focused less on designing hardware and code from scratch and more on combining and modifying existing code and libraries to function in an embedded environment. After evaluating our team members' educational backgrounds and practical experience, we decided that we were best suited towards one of these projects. So the real choice lay in what game to port.

For a game to be suitable for our capstone it had to be open-source, multiplayer, and from a time period around that of Quake (1996) or earlier, as more modern games require performance levels that would be difficult to achieve on the Virtex 5 series FPGA. It would also be preferable if the game was built in such a way as to easily support porting to a new platform. With these criteria in mind, we decided to use Doom, which was originally released in 1993, has networked multiplayer, and has been ported to many platforms after its open-source release in 1997.

Game Details

After researching the various Doom ports available, we settled on a port known as Chocolate Doom. Chocolate Doom aims to reproduce the original Doom as closely as possible, right down to all the original bugs. More importantly, Chocolate Doom is built using Simple DirectMedia Layer (SDL), a popular cross-platform multimedia library designed to provide fast access to input/output devices like audio and video. SDL abstracts away the platform specific details as a small set of device drivers, then calls on those drivers for the rest of its functionality. SDL is particularly careful to separate all of its hardware-specific functionality from the rest of its code, such that SDL can be ported to a new platform simply by providing a very small set of drivers

and setting SDL's configuration to point to those drivers. Chocolate Doom's reliance on SDL made it a particularly suitable candidate for our capstone project.

One potential complication was that Doom uses a single data file, known as the WAD (standing for Where's All the Data?) to aggregate all of the data needed by the Chocolate Doom executable. Although this file conveniently lumps all of the sounds, sprites, and configuration information into one file, it also meant that we had to create file-system support as well. In addition the file-system support would have to be implemented very early on for Doom to function correctly, which would be necessary for testing the various drivers we wrote.

Approach

Our general approach was to first familiarize ourselves as much as possible with Xilinx's complex tools and available IP, as well as Doom and the SDL libraries. Once this was accomplished, we needed to build a hardware platform capable of running the Doom executable. To do this we would have to incorporate basic hardware such as a processor and memory, as well as I/O controllers for audio, video, keyboard, networking, and Compact Flash. The next layer in our design was to be composed of the basic libraries provided by Xilinx such as the standard C library, threading, etc. On top of this basic software platform we needed to implement drivers for SDL to interface with our hardware and Xilinx's standard libraries. Finally we would need to make changes to the Doom and SDL source code in order for it to compile correctly under the Xilinx toolchain.

Division of Labor

Only one of our three team members had any significant previous experience with computer architecture and computer programming, while the other two had more experience with hardware design. Thus the most natural assignments seemed to be assigning the software jobs to that team member (Rob) and assigning the hardware design to the other two. Fortunately Xilinx's XPS toolchain provided a very distinct boundary between hardware and software. Hardware could be designed and compiled in XPS, and the completed design could be exported to the Xilinx SDK. The SDK divided projects into three distinct components – hardware designs imported from XPS, board support packages composed of drivers and libraries targeted toward a specific hardware design, and user applications. This meant it was very easy for our team to conduct work on hardware and software concurrently while occasionally exporting the newest hardware designs to SDK.

For the first part of our project, the hardware and software work could be done almost completely independently, as the first step for software was to simply get the Doom and SDL code to compile under the Xilinx tool-chain without regard for the actual hardware. The second step for software however was to build drivers for our hardware design, which meant that each piece of hardware had to be added and the new design exported to SDK before the relevant drivers could be written and tested. This required a little more coordination between those developing the software and hardware, although it was by no means unmanageable.

Scheduling

After determining the division of labor we laid out a schedule that was roughly divided into two parts. The first half of the semester consisted of a week to install and familiarize ourselves with Xilinx's tools, followed by roughly two weeks of performing example tutorials provided by Xilinx. The remainder of the first half was used on the software side to get Doom and the various SDL libraries to compile correctly, and to develop the basic platform on the hardware side. We allocated more time for this than we were initially inclined due to warnings from previous reports that this stage was much harder (and more frustrating) than they thought. The next stage was composed of adding hardware pieces to the basic platform while simultaneously adding their software drivers to SDL, at the rate of about one per week. The final week of our project was allocated for testing and also served as our "buffer" in case we ran into trouble and needed more time to complete our project.

We ended up being consistently about a week behind schedule, which was okay because we had built in a buffer to allow that. For the mid-semester demonstration we were able to show the Doom and SDL code as a fully compiled ELF under the Xilinx toolchain and basic audio, video, and keyboard functionality. Just before Thanksgiving break we ended up falling much further behind however, mainly due to difficulties implementing the SDL drivers. Two of our teammates stayed at Carnegie Mellon for Thanksgiving break however and managed to catch up in time for the demos.

Hardware Design

Our hardware design consisted of a processor communicating with DDR-SDRAM, an AC'97 sound controller, DVI video controller, PS/2 keyboard controller, Compact Flash controller, timer, RS-232 controller, interrupt controller, and Ethernet controller. These components communicate over two PLB buses, with the few high-bandwidth components (memory, AC97, and DVI) on one bus and the other, slower components on the other. In addition we had to add an OPB bus, as the AC97 controller we were using was a reference design from several years ago and was no longer supported by the current Xilinx tool-chain. Not only did we have to manually add the AC97 IP core to the Xilinx library, but we had to add an OPB bus solely for the AC97 controller and connect it to the PLB using a bridge, as it did not have a PLB interface. The timer and interrupt controllers were central to the proper operation of Xilkernel (discussed below) as the interrupt controller could aggregate all the different device interrupts and deliver a single interrupt to the Microblaze, while the timer provided the constant interrupt necessary for thread preemption.

Our hardware design needed to run our game at a high enough rate for it to be playable, and in order to do so we looked to tailor the design to suit the software. We decided to choose a soft Microblaze processor, as this would allow us maximum flexibility in our hardware design for what we believed would be a minimal loss of performance compared to a hard-wired PowerPC core found on some of the other boards. We also made sure to enable as many performance-oriented features of the Microblaze as possible such as caching and hardware floating-point support while removing unnecessary components such as exceptions. Although these preliminary design decisions resulted in a game that was playable, it was still not running at the frame-rate we were hoping for. We were left trying to come up with new and creative ways to

squeeze more performance out of AC97 with only a few days left before the public demo. Although our hardware team tried many different approaches, ranging from increasing the clock frequency of various peripherals to removing unnecessary components to exploring direct-memory access, we simply did not have enough time to squeeze any more power out of the design. We did come up with at least one significantly faster design, but this design involved a rather unstable increase to the DRAM clock frequency which seemed to cause occasional lockups and was deemed unsuitable for the final demo.

It should also be noted that the specifics of the software application helped guide our hardware designs and saved us from exploring many fruitless methods of increasing performance. For example, we were seriously exploring using a direct memory access (DMA) controller before we realized that the software application simply did not perform the large memory transfers for which DMA is optimized. Similarly our debugging efforts on the software side when testing the software drivers led us to the unexpected conclusion that memory accesses were simply taking too long, which led us to discover that caching had not been enabled properly! After fixing the problem our application ran much faster, thus saving us from what could have been a very long and frustrating search for a software bug that didn't exist. It just goes to show how important it is to have a complete understanding of both hardware and software when designing a complete embedded system.

Software Design

The software portion of the project was able to be conducted almost entirely independent of the hardware part for the first half of the semester, mostly because the first major task was to get the software application compiling under Xilinx's toolchain. The software project was composed of four major parts: the Chocolate Doom source code, the base SDL library, the SDL-mixer library used for audio processing, and the SDL-net library used for networking. The first step was to import all of the relevant files into an SDK project targeted towards our hardware and get it to compile into a working ELF file. Thus began a very long and frustrating cycle of attempting to compile, reading the resulting error messages, examining the indicated course code, modifying it to fix the error, and attempting to recompile once again. The most common errors were path errors (an #include directive pointing to a no-longer valid path), namespace errors (global identifiers that were used in more than one of the four aforementioned components), standard library errors (standard function calls that were not supported by Xilinx or replaced by functions with slightly different names) and missing component errors (some of the files we imported were meant to be used on different systems and thus would not compile, which we fixed by excluding them from the build). We were able to get a compiling ELF mostly due to SDL's null drivers, which allowed us to select an empty set of drivers instead of one configured for a particular platform. This allowed us to get a compiled ELF before we had to actual worry about implementing any drivers and served as a good launching point for our own implementations. The compilation step took longer than we originally anticipated but fortunately not longer than we had scheduled.

Once we had an ELF that could be downloaded to the FPGA and run, the next step was to actually implement the empty functionality. The first step was to choose a board support package that would provide the underlying drivers and standard libraries for our C code, with

the two choices being Standalone and Xilkernel. We chose the latter, a simple and well-documented operating system developed by Xilinx to run specifically on their boards which provided many useful functions such as Posix threads and interrupt handling (which saved us a lot of time).

The next and perhaps most important step was to get the filesystem working, as Doom reads the WAD file almost immediately on startup and then reads and writes several temporary files during execution. Without the data provided by those files, execution of Doom simply couldn't proceed far enough for us to test any major functions such as keyboard input or video output. The complication was that although Xilinx's standard C library provided the standard fopen, fwrite, etc these functions are null implementations that don't actually do anything (a fact that was not made very obvious by Xilinx's documentation). Instead once the user chooses a file system they must use a set of functions whose names are unique to that file system. Furthermore the file systems provided to us often had limited functionality (for example, a read operation exhibited the same behavior for end-of-file and error conditions). Finally we had to contend with the fact that the Doom WAD would have to be stored on Compact Flash, which was too slow to be used during execution of the application.

Our solution was to implement a file-system wrapper we labeled "XFS" for "Xilinx File System." XFS was composed of a large collection of wrappers which A) provided a mechanism to load several files from Compact Flash to an in-memory file-system on startup and then B) provided a set of wrappers to the memory file-system that were compatible with standard functions. For example, if the prototype of the standard function "fopen" is "FILE * fopen(const char * filename, const char * mode)" then XFS provides a matching function "FILE * xfs_fopen(const char * filename, const char * mode)" whose behavior mirrored that of "fopen."This made it very easy to port over the file-system functions: we simply replaced all instances of "stdio.h" with "xfs_stdio.h" and then changed all calls of standard function "f" to calls of function "xfs_f".

Once this hurdle was overcome the next step was to implement the SDL drivers for the file-system, timer, threading, keyboard input, audio output, and video output. Implementing the file-system, timer, and threading drivers was fairly easy as our XFS functions and the timer/threading functions provided by Xilkernel were mostly Posix-compliant, which allowed us to reuse most of the Unix SDL driver code. We then spent a significant amount of time reading the SDL source code in order to understand it better before tackling the remaining, much more complex drivers. Audio however proved to be fairly troublesome, as once the implementation was completed the audio was jerky and unintelligible. Thus began a very long and arduous attempt to debug the audio driver which finally resulted in the surprising conclusion that the driver code was functioning correctly, but our memory accesses were taking far too long. This led us to discover that caching was not enabled correctly, and after that fiasco development on the remaining drivers proceeded much more rapidly.

Development of the video driver was fairly straightforward, although we did later discover a bug where Xilinx had labeled control bits for a particular register with the MSB as 0 and one of us had interpreted the bits with the MSB as 31 (got to pay attention to those little details!). The keyboard driver was also uneventful although a bit tedious due to a need to write a table translating all possible keyboard scan-codes to SDL keys. The next major hurdle occurred when

we ran the game and discovered an error when SDL attempted to play a file containing background music. The reason was that Doom encodes all of its background music as MIDI files, and in order to play MIDI files SDL requires either native hardware support or an entirely separate library previously unknown to us known as Timidity.

Unfortunately we quickly discovered, after wasting the entire night before the public demo porting Timidity to our project, that Timidity had been developed with much more modern computers in mind than those that ran Doom, and even by those standards it was considered processor-intensive. This meant that enabling the Timidity library brought our application to a screeching halt due to the overwhelming load – although for all intents and purposes it appeared that the library was working, the on-boot pre-processing by Timidity alone took over an hour just to get through half the instrument files. We finally concluded that Timidity was a lost cause and would never be able to run on such a slow system, and subsequently disabled it for the public demo.

The last component we wanted to include was networking. This should have been fairly straightforward as Xilinx provides the "lwip" network library for the software application, which in turn provides a Posix-compliant sockets interface that was fairly easy to use in our SDL networking driver. Unfortunately Xilinx would not allow the user to include certain critical sections of the lwip library in the Board Support Package build process, and attempts to manually override this in both the configuration files and compiler options were in turn overridden by the SDK. After several frustrating hours attempting to get around this problem, we decided to simply copy the lwip source code to our own project where we would have more control over it. This was in turn followed by a several more hours spent tweaking the ported code so it would compile correctly in our environment. Although we did finally get all of the networking code successfully compiling, we never got a chance to actually test it; although it theoretically should work, this was unfortunately not confirmed in time for the public demo.

Final Product

Our final product was a CompactFlash card containing a SystemAce that, when inserted into a LX110T FPGA board, would configure the board on boot with our selected hardware, initialize the hardware, file-system, and operating system, copy all the required files from the Compact Flash card to main memory, and begin executing Chocolate Doom. The user could view the original Doom game on an attached VGA monitor, hear foreground sounds (such as gunshots and yelling, but not background music) from the attached speakers, and control the in-game character using the attached PS/2 keyboard. Although the frame-rate was not what we had hoped and there was a noticeable lag for player commands, it was certainly still playable and our team members spent a significant amount of time playing the game ourselves. Despite our inability to include fully-functional networking in the final product, we believe that we met our most important goals and were proud to having a working product available for the final demo.

What We Wish We'd Known

It would have saved us a lot of time if we had known Timidity was a lost cause from the beginning instead of discovering it the night before our final demo. It was unfortunate that all

the background music files were in MIDI format, that Timidity was the only option in SDL-mixer that would work for our environment, and that Timidity was simply incapable of functioning on our hardware. If we had understood this at the beginning, we might have been able to find a source port of Doom that did not use SDL while still maintaining its portability, or perhaps a version that used WAV files for the background music as well as the foreground.

We also did not know going into this project how difficult it would be to debug other people's code. When you are working on a project that was designed and implemented by you and your team from the ground up, it is much easier to find an error due to your intimate knowledge of the project. For our project however we were trying to integrate several large, complex software libraries from multiple different vendors into one cohesive whole, on top of a hardware design that was similarly composed of IP cores supplied by Xilinx. We spent far more time reading through mind-numbing amounts of documentation and code than we did actually writing anything.

Advice for Future Classes

It's very important to choose a project that plays to your group's strengths. Hopefully your group will have a diverse and valuable set of skills, but if your group is lacking in certain areas then you should choose your project accordingly. On a related note, don't take the prerequisites lightly. They exist for a reason. Our group ran into a lot of trouble because only one of our group members had taken more than one of the prerequisites, whereas the other two were lacking the basic education about computer architecture and computer systems necessary for this course.

Also, make sure your group communicates effectively. There were a couple of points throughout the semester where better communication might have resulted in a more efficient use of labor, as misunderstandings can result in an incorrect implementation, or one team member wasting time trying to solve a problem that was already encountered and solved previously. We used a Google Group to provide a central forum for updates, file sharing, and scheduling meetings which was very helpful.

Make sure you invest some time at the beginning of the class getting set up and familiar with your workspace before diving right into your project. Set up your favorite OS, find and download any bugs fixes for the tools you are using, and familiarize yourself with Xilinx's insanely complicated toolsets. Our team took a week or two at the beginning of the semester to set up a Linux workstation for software development, work out compatibility issues between Xilinx and our board, and run through every XPS and SDK tutorial we could find. Although running through the tutorials was grating, the knowledge we gained proved immensely useful throughout the rest of the semester and kept us from running into a myriad of problems down the line.

Take the time to build a comprehensive testing framework to help you with your debugging efforts later in the semester. Most engineers are far more interested in building something than testing it ("I know it works, I don't need to prove it!") but the time invested will pay off dramatically by allowing you to quickly test your project after you make any major changes.

Catching new bugs early will save you valuable time that you'll almost certainly need when you are awake at 5am the night before the demo desperately trying to finish your capstone!

Finally, read the documentation!! This was especially important for our project as we were using large amounts of code and IP cores designed by other people, and we needed to understand all the details of their operation to effectively integrate them. We had several bugs that could have been prevented if we had read the documentation thoroughly the first time, instead of skimming over what we didn't think was important. It can be difficult to force oneself to read a forty-page technical document without skimming over it, but the minding-numbing reading now is certainly worth avoiding hours or even days of frustration and wasted time looking for a bug later.

Personal Notes for Rob Waaser

From the beginning it was pretty obvious that I was going to have a difficult time in this class. I was the only one who had taken more than one of the prerequisites for the class, and the only one with significant experience in computer architecture or computer programming. Ironically enough I was also the one with heaviest workload; as I was pursuing a dual-degree in computer science and electrical/computer engineering, I had been forced into taking four additional technical classes this semester, three of them project based, in order to graduate on time. Needless to say I never, ever would have done that if I'd had a choice in the matter, but that's life.

I seemed to be the most comfortable with large software applications so I volunteered to handle the software side while Shupeng and Soo Hwan handled the hardware design. The first thing I did was to install SDL, SDL-mixer, SDL-net, and Chocolate Doom on a spare Linux computer I had lying around at home. In retrospect this was a really good idea because I knew exactly what our final product should look like at the end of the semester. This let me know what performance and graphic detail to expect from a properly executing Doom game and let me catch bugs such as when a bit numbering error caused our greens and reds to be switched on the video monitor.

My first job after that was to convert one of our workstations to run Linux and install the Xilinx EDK on it. It took a few days to work out the compatibility issues, but in the end it was definitely worth it due to the fact that I am much more comfortable developing on a Unix system than Windows, and I am more familiar with "productivity utilities" such as "grep." After this was done I took some extra time to explore SDK, writing some simple device drivers and "hello world" programs to run on our board. After I was satisfied that I understood the SDK, I began the arduous task of importing all of the necessary source code for our project and massaging it to compile under the Xilinx toolchain. This was especially frustrating towards the end when I started running into inexplicable linking errors telling me that implementation were missing for some of the standard library functions such as "link" and "unlink." After wasting several days trying to fix these errors, I finally just created some null functions to provide these implementations after determining that Doom doesn't even use them in the first place!!

Things got much worse a few weeks into the semester, when I was unexpectedly informed that my fraternity was having its charter revoked and that the residents of the fraternity house were to be evicted within a month. I was thus sent scrambling (along with about thirty other fraternity members) to find housing for the rest of the year when most of the available housing had already been taken at the beginning of the semester. Thus there was a period of several weeks where my participation in this class dropped drastically as I struggled to simultaneously finish my schoolwork, search for housing ads, visit houses, fill out rental applications, go to job interviews, and attend meetings with university officials. Fortunately we convinced the administration to let us stay in the ex-fraternity house until the end of the semester, but the pressure of finding housing by then continued to follow me until I finally acquired a place after Thanksgiving break.

Once I got a little breathing room my next task was to implement the XFS wrappers discussed above. What was probably the most annoying about this was that I couldn't implement all of the functionality for the Posix functions I was trying to mirror because the underlying file-system I was using, Xilinx's Memory File System, simply did not provide enough functionality itself. For example, there was really no way to differentiate between a read error and end-of-file. I tried to implement as much Posix functionality as I could but had to leave a lot of the "error handling" unfinished, which ended up biting me later when it turned out to be the source of a subtle bug deep in Doom's source code involving a file read followed by a check of the error code returned. However for the most part the XFS wrapper worked beautifully after I converted all of the relevant function calls to use XFS instead of the standard library.

As Thanksgiving approached I became increasingly uncomfortable with where we were, as almost no progress had been made on the SDL device drivers at that point. Since I had just finished the file-system wrapper, I elected to stay over Thanksgiving break and crank out the SDL drivers myself. As this was the first time I had really been able to sit down and concentrate on this project without worrying about my other classes, I was able to make quite a lot of progress. I spent every waking hour in lab working on the code, and by the time the break was over I had completed and successfully tested the timer, threading, file-system, audio, video, and keyboard drivers.

It didn't take much effort from there to get Doom working, and by the time the in-class demo rolled around we were able to demonstrate a playable version of Doom, albeit at a nightmarishly slow frame-rate. After systematically testing and evaluating my code, I finally came to the conclusion that the memory accesses themselves were taking far longer than they should have. The most likely reason was that caching wasn't working properly, which was surprising because we thought we had enabled it a while back. After discussing the possibility with Shupeng, he discovered information which led us to believe we had misinterpreted the Xilinx documentation about how to enable caching. Shupeng fixed the problem, we ran the application again, and immediately we noticed a significant boost in performance (especially for audio, which had been previously incomprehensible).

As Shupeng and Soo Hwan continued to work on improving the performance of our project, I worked on completing the last two requirements for our project – networking and background audio. Audio was particularly frustrating, as I discovered less than a day before the demo that

SDL-mixer was lacking an entire library to play the MIDI files used for Doom's background music. After wasting most of the night porting the library over to our project it became overwhelmingly apparent that the library (Timidity) was simply too CPU-intensive for our humble hardware design to handle. I had to manually disable background sounds before I went to work on the networking port, and due to the wasted time was unable to test networking in time for the demo.

Overall I thought this class provided me with a lot of valuable experience. I liked the emphasis of this class on the project instead of things like homework assignments, etc. I also liked the attitude of "make it work" – it didn't matter how you got your final product working, even if you used other people's code (as long as it was properly cited). I feel like this is a much more accurate reflection of "real-life engineering" than most of the other classes I've had at CMU. Furthermore my understanding of the hardware-software interaction layer was dramatically improved, since despite taking embedded systems classes and operating system classes I had never written a driver for anything more complicated than a timer. In fact, I am very appreciative of the practical experience and understanding I gained about embedded systems in general, as I never would have picked up some of the things I learned without tackling such a large project. I was also able to significantly improve my debugging skills and my "massaging" skills (tweaking things so the compiler will accept them), because I swear that's 30% of what I did this semester.

Personal notes for Shupeng

The most important task for me was to build a robust and complete hardware infrastructure which is capable of running Doom. At first glance, I thought it was an easy task. When throwing myself into work, I realized the journey was filled with obstacles, and some unexpected errors always made me crazy and forced me to sit in the lab for several hours just for one bug. More time working on it, more difficult I found it was. Though challenging, it was rewarding. I was happy for every tiny progress. Step by step, I finally accomplished our goal: a robust and complete hardware platform.

Hardware Design

According to the requirements of our project and the characteristics of our board, we made a block diagram for our hardware platform. It included processor, memory controller, buses, timer controller, DVI controller, AC'97 controller, PS2 controller, and etc. I chose Xilinx Platform Studio (XPS) to build our hardware project. There exist many tutorials on the Xilinx website, so getting started is not hard. During the initial stage, most of time was spent on coordinating the compatibility of XPS and our board. XPS 12.2 doesn't have device library for XUPV5-LX110T. To solve this, I downloaded an expansion package. The most difficult part when applying XPS is how to make connections between different ports, how to set the parameters for different IP cores, and how to distribute addresses. To get over these difficulties, the user needs to be extremely familiar with FPGA and computer architecture. I seldom touched hardware architecture before this course, so it was really a big challenge. Most tutorials from Xilinx are focused on building a basic design which is helpful for the beginners. As for how to build a complex infrastructure like ours, they don't mention. After some search, I found some tutorials

talking about how to add AC'97 on the basic design, or how to add DVI on the basic design. I carefully pieced them up and tested the accuracy after adding a new component every time. Compiling hardware design and generating bit stream in XPS usually took about an hour. Each stupid error would make this hour wasted, and I needed to start again. I cannot remember how many hours I sat there just for adding one simple component. Some errors in XPS were weird, and sometimes I even had no idea what XPS was talking about. If I didn't know what was wrong, how could I fix it? Could Xilinx provide us with a more friendly development platform!? Hours after hours debugging, this was my life. Of course, I learned a lot during this process. From those stupid errors, I can easily figure out where is wrong and how to solve it. Time pays!

Hardware Test

Passing the hardware compiling does not mean your hardware can work correctly. Different hardware settings can lead to different hardware infrastructure. So we need to test whether it works as we thought. Each time I added a new component to our existing hardware project, I would write simple applications to test its accuracy. For some components, it was easy to write test applications and sometimes I even luckily found some test applications from Xilinx website. For example, when testing DVI, I could use some functions existing in the ISE library because DVI IP core has been included in ISE. However, for some components, testing consumed lots of time. For example, when testing AC'97, I had to write drivers myself. Initializing AC'97, reading and playing audio files all touched registers and buffers. Old labs provided us with some reference codes. I modified these codes and wrote drivers for our test. At first, it didn't work. I thought the hardware was not correct. While after several weeks' check, I finally found out that our software application compiling environment hadn't been set correctly. Compiling environment setting was very easy to be ignored. So one advice I can give now is double check the hardware settings, software compiling settings whenever you meet errors. Running applications on FPGA relates to many parts, and one wrong operation may kill your project. Another suggestion I want to mention is documentations always help. When testing PS2 keyboard, I met with an error. The application worked correctly when I pressed one key at a time. While pressing several keys together or holding down keys without release, the application failed. Then I read the documentation and found out the solution. When you press a key, a scan code called 'make' code will be transmitted to the processor. When you release a key, a scan code called 'break' code will be transmitted. The make codes and break codes are stored in a FIFO buffer. When holding down a key without release, the make code will be sent continuously. The break code will not be sent until you release this key. If you understand how it works, it is not difficult to use it. Just as my teammate Rob said, read the documentation. I have to say they always help.

Cache and FPU

We didn't consider adding cache until at the last moment. When running Doom on FPGA, we found that it was extremely slow. One video game looked like slides! Rob told me that it was because Microblaze was not fast enough to process so much data. One way to improve the speed is to use cache. At first, I thought cache was built in memory itself. If we wanted to use cache for DRAM, we should distribute certain address and build a cache in DRAM itself. After test, we found that the performance was not improved in this way. After further study, I found that cache was built in the processor using BRAM. So if we wanted to use cache for DRAM, we

should enable the cache in the Microblaze and provide certain cache connection ports in DRAM. Subject to the number of BRAM, Microblaze only allows us to use at most 64KB cache. After test, the performance was improved dramatically. However, the speed was still not fast enough to run a video game. Then I enabled Float Point Unit (FPU) in Microblaze, the performance was further improved.

How much frequency can we get by using Microblaze?

In the last few days, I was trying to build a faster processor. After enabling cache and FPU in the Microblaze, the only way I could think of was to improve the Microblaze frequency. The maximal frequency we can use in XPS is 125MHz. This frequency is conservative to guarantee all the components in FPGA can work correctly. From the Xilinx Forum, one Xilinx engineer said that Microblaze frequency could be increased to approximately 250MHz. Of course, we should consider the tradeoff of frequency and performance. When performance is high, Microblaze would use more BRAM and frequency will correspondingly decrease because of the delay passing these BRAM. Another thing we should consider is that different frequencies used in the hardware are not independent. Some components have to use the same frequency as that used by the bus, and some components have to use the double frequency. So we are not allowed to choose an arbitrary frequency for each component. To generate different frequencies, we can use clock generator in XPS. I carefully increased Microblaze frequency from 125MHz to 200MHz. With some other modifications, I passed the compiling and generated the bit stream. However, when doing the test, Microblaze refused to work. Then I decreased the frequency from 200MHz to 150MHz. No response either. I guessed there were some strict constrains for Microblaze frequency in XPS. When the frequency is higher than the limit (125MHz), the hardware is locked. As it impossible to increase the Microblaze frequency, I tried to improve the frequency for DRAM. I increased the frequency from 125MHz to 250MHz. The performance was improved correspondingly.

Generate ACE File

During the development stage, we used host PC to compile and download the configuration file (bit stream) and executable file to FPGA. Configuration file is used for hardware configuration. Executable file contains all the application information. During the game testing stage, we didn't need host PC anymore. I packaged configuration file and executable file into an .ace file using Xilinx bash. Then I copied it into Compact Flash card. Now we could configure our game using Compact Flash card. This made it possible that we gave a public demo with only a FPGA board, a monitor and a PS2 keyboard.

Summary

We finally presented a playable Doom game. I am really proud of our work. From the beginning, we discussed our project and made a very clear goal. Then we followed our schedule and moved forward step by step. Yes, we met with many difficulties. We were frustrated sometimes. But we always knew we didn't fight alone. We fixed kinds of problems by joint efforts. Through 18545, I know better about computer architecture. Though hardware always drove me crazy, I

find it interesting now. Seeing my idea running on FPGA is really cool. I appreciate this experience.

Personal Note for Soo Hwan Kim

In Team XilDoom, I was responsible for setting and building the hardware infrastructure needed in running the Doom game source code. As Rob said he would take care of the software himself, Shupeng and I basically formed a 'hardware team' in which we would either work together in the lab or carry on from where each other left off alone. The entire process that took place from September to December made out to be a happy ending as we were able to run a playable demo of Doom on the FPGA board (although the frame rate was not as fast as we anticipated). However, it is quite amazing how much time I spent in learning the configurations, reading and finding the right documentation, and controlling my frustration. Sometimes, I would sit in the lab from 11 a.m. to 8 p.m. only to find a single stupid bug that I would have found out easily and earlier if the Xilinx had documented well. In the end, we had a working product and a playable demo ready for the public show, so I was happy. As I am graduating this semester, I feel all the times I spent in the lab and at home doing the project will become a good memory to look back on time to time.

The Beginning

Building a hardware product and setting it accordingly, as Shupeng often said on a chair besides me, was a 'disaster'. We chose the Xilinx Platform Studio (from now on, simply XPS) as our software in building the hardware, and our struggle through FPGA started when the latest XPS did not contain a device library for our board, XUPV5-LX110T. After some time spent in searching, we found out that there was an expansion package that had the library we needed. All three of us, as a team, first gathered and discussed how we should design our project. We decided that the major hardware components that we need for running Doom were the DVI controller for video, AC'97 for sound, PS2 keyboard for input, and Ethernet for network. In the end, we were able to successfully configure the DVI, AC'97, and PS2 keyboard hardware. Shupeng finished building the Ethernet on the board, but as he faced problems in testing it, we never know whether it worked or not.

Most of the first few weeks of the semester were spent in learning how to use the XPS and our FPGA board itself. I had not taken 18-447, so I was not too familiar with computer architecture nor the in depth hardware design, so fair amount of time had been spent in reading the documentation of each and every part of the board, and many other reports in the website. Particularly helpful were the old lab reports, especially Quake and Descent, as we they had similar approach to their project as we did, and therefore, we found many overlapping errors or obstacles along the way that were very similar or the same. Thanks to their well-done documentation of the report, I was able to sometimes find our ways through faster from their report and not the Xilinx Forum or Google.

One thing that I really liked about Xilinx is their tutorials to setting up a basic hardware component. I used the tutorials for the DVI, AC'97, and the keyboard to provide a basic

hardware for each, and they were of good help as I was able to grasp a basic understanding of what's going on around the board. Sometimes, Shupeng had already finished configuring the hardware, but I would come back to the lab at night and do the entire tutorial alone all over again in order to learn it myself.

AC'97

The hardware for AC'97 was the one that took the most time in building and therefore the one that provided me with much knowledge of the Virtex-5 board. Building AC'97 on to the board was not a difficult process, but it certainly needed me to test my patience and perseverance as the entire process of compiling and generating bit streams took about an hour to complete. Unlike many of the other programming projects that I have done, it was actually better to thoroughly look at every piece of hardware, the configurations, and the connections of the ports, before compiling rather than after I found about the problems. Compiling without much thinking or without careful thinking would definitely mean bugs in the process, and it was a lot better and time efficient to spend maybe half an hour or an hour to do everything carefully and thoroughly than compile for an hour, find bug, correct, compile again for an hour, find bug, correct, and compile again for an hour. It was this process of building AC'97 on to the board that taught me the importance of thoroughness in this course.

There were, of course, many other problems in building the AC'97. One of the problems is the overlapping pin with the USB port. Initially, Shupeng and I decided to configure the USB port rather than the PS2 inputs, so we downloaded the tutorial for the USB and followed every instructions in it. However, as Xilinx just provided us with a basic tutorial for setting up each hardware component individually, we faced a problem during the configuring process where the one of the pin used in setting up the USB overlapped with the pin used for building the AC'97. This, if we had more knowledge of computer architecture back then, would have been able to be solved, but as we had not much understanding of the board at the moment, decided to abandon the USB and move along with PS2.

Even after successfully building AC'97, I faced other problems. This was later found out to be an extremely stupid thing, but all I had to do was enable the use of AC'97 within the XPS. The step that only needed a few clicks in XPS made me to spend three whole days in the lab making me do the entire process over and over until I finally found out that there was no problem with the setup I had done just that I had to simply 'enable' it. Even until yesterday, whenever I thought about those three days, it drove me nuts, but now that I think about it after carefully outlining my progress in the report, I think this was the part that made me learn about our project. If something cannot be understood, a simple repetition will solve that problem.

PS2 Keyboard

PS2 keyboard was another piece of hardware that taught me with so much pain that if I read the documentation more carefully, everything will be done much more smoothly and much faster. Actually, this is the thing that Rob has kept on telling me throughout the course that I refused to listen until the very end. Although I knew it was better to read the documentation thoroughly and have more in-depth understanding of the hardware before configuring them,

due to my personality of always wanting to finish things fast, it usually put me in situations where I would have to spend more time than it would have normally taken in finishing a project. This was especially the case in this course.
At first, I did not know where or how to start configuring the PS2 keyboard, and I was later shocked to find how simple it was. Maybe, I was expecting more of a challenge as the process of building AC'97 on the board was like a hell to me. The problem was that Shupeng and I took turns in building the PS2 keyboard on to the board, but we had no idea how to write a program to test it. After searching many documentations here and there including the Xilinx Forum and Google, we found a very good website that basically outlined the basic set up and how the PS2 keyboard interacts with the user. Shupeng was the one to first understand how it works and told me how simple it was. It turns out that for each key on the keyboard, there is a value for each time it gets pressed and released. So in the process of typing the character 'a', there are total of two values being outputted: one, the value that tells me the character 'a' has been pressed, and two, the value that tells me the character 'a' has been released. Therefore, if I pressed 'a' and keep it pressed for a long time, it will only output the value corresponding to it being pressed for a long time and not output the value for the release until I finally release it. After having complete understanding of how PS2 keyboard works, writing out a test driver was really easy. A few nested if-and-else loops were all we needed, and I was shocked to find this out as I spent much time trying to write a driver that took less than few minutes only if I had read the documentation more thoroughly. This was when I really felt the importance of reading the documentation.

Combining

Other problems I faced with the AC'97 and the PS2 keyboard was the process of combining the two individual projects to a one single project. At first, what we had were two separate projects for AC'97 and the PS2 keyboard that were only capable of running each hardware piece individually. However, for our game, we cannot have them run separately but synchronously on one single project. It took time, but we finally managed to combine the two separate projects together into a single one, then we also added in the project for DVI, and a single project capable of handling the video, audio, and input were finished. Hey, all that was needed in running the game!

SDL

Another huge problem I faced was the SDL. SDL is a crucial part of our project. It acts as a bridge between the game and the hardware we implemented. How it works, simply put, is that the game calls the SDL, and then it calls the hardware we implemented to complete the software-hardware connection. Shupeng and I were responsible for writing out the drivers for the hardware pieces we implemented, but even after spending several weeks trying to learn how to write a simple driver, I was not able to write a successfully working SDL driver. I asked for help to the TA, earned some advice from the professor, including emailing the previous students who took 18-545 and used SDL in their projects for help. However, they all ended up in failure as the TA was not familiar with the SDL, and the guys who took this course last year refused to answer our email. Even after weeks spent in trying to write a simple driver for the SDL, I ended up not being able to. However, it was then Rob came in and wrote the entire drivers including the

video, sound, keyboard, and network himself. I have to admit. He is amazing, and I still do not know how he managed to understand and write the drivers, but thanks to him, our big problem was solved.

Performance

After Rob finally downloaded the doom source code on to the FPGA board successfully, we were finally able to test out the game. I have to admit. I kind of hoped that it would successfully work on the first run and anticipated for it. Quite amazingly, it turns out that it actually did! However, we immediately found out about a problem as the game was running in very slow frame rate. We managed to get the game with the poor performance running in the very last week, and unfortunately, I had two midterm exams that week, and I was unable to help them solve this problem right away. When I finished writing the exams, my hardware partner, Shupeng, had already found the way to fix this problem. It turns out that he enabled the DRAM cache and the Floating Point Unit (FPU) in the Microblaze processor, and this made the game run at a much faster frame rate. However, the game was still not fast enough to run at its original frame rate, and I had to find ways to solve this in the last day.

In the end...

Although it would have been great if I had found other ways to improve its performance and make the game faster so that it runs at its original frame rate, I was not able to find any other ways. For the whole day, I sat down in front of a computer reading the entire Xilinx Forum about increasing the speed of Microblaze processor. I found some useful answers, and told Shupeng, who was fiddling with XPS, to apply these advices, yet we found out in the end none of them were successful. We tried removing unnecessary hardware pieces, manually set the clock frequency of the Microblaze from 125 MHz to somewhere over 200 MHz, but with the reason still unfound and unknown manually setting the clock frequency to over 200 MHz did not quite work. The Xilinx employee answered in the Forum that the maximum clock frequency rate the Microblaze can provide is 250 MHz, and there were guys who took the advice, manually set the clock frequency to somewhere a little below the maximum, and reported that it worked, so we tried doing the same, but it did not quite work.

Also, I would have liked it if Xilinx were more detail in their explanation or advices or even provide some *useful* advices. One of the answers I found from their employees telling one of the guys who asked a question in how to speed up the Microblaze process using the Virtex-5 board was to 'purchase a much faster Virtex-6 board'. I mean, yeah, that could really speed things up, but that was not the kind of solutions we were looking for. Shupeng and I tried numerous more things including removal of unnecessary hardware, increasing clock speeds of individual components, and others, but we were not able to find any noticeable performance improvement.

Summary

This course was sort of a new experience for me because I had not much knowledge of computer architecture unlike many of my classmates who did. Basically, they were applying

their knowledge from the start which was very different from me because I had to learn along the way and then apply what I learned immediately. It took hours and hours during the process, sometimes with much frustration and anger, but many hours spent in trying to learn each and each individual component finally paid off as I was able to grasp more basic understanding of the hardware and its surroundings. This course offered me a chance to explore a new area of hardware design, the infrastructure, and the implementation. Although, during the very first few weeks of this course, I was keep wondering 'why am I doing this? Why am I taking this course?', as time passed on, as I actually got more interest in hardware design and FPGA boards in general. One thing was for sure. Endure and persevere. They will pay off. It was the first real-life corporate kind of experience that I had in my life. I really enjoyed it.

References

Information on SDL: http://www.libsdl.org/
Information on Chocolate Doom: http://www.chocolate-doom.org/wiki/index.php/Chocolate_Doom
Information on Doom's internals: http://doom.wikia.com/wiki/Entryway
Lots and lots of Xilinx IP datasheets, tutorials, and other documentation