XiQuake II

Benson Tsai

Alfred Barnat

Aaron Mintz

We ported Quake II by id Software onto the Virtex 5FX FPGA complete with a playable single player and over the network multiplayer. User input through keyboard and mouse and the graphics is output via the DVI controller. Networking is also done through on-board ethernet. As a base foundation, we used Linuxlink by Timesys which is a Linux distribution catered toward embedded platforms such as the Virtex 5FX. SDL libraries were used to provide graphics and input support.

Who are Thou?

Quake II (1997) is a first-person shooting game made by id Software. Like its predecessor, Quake, the Quake II engine (id Tech 2) was open sourced by John Carmack under the GNU General Public License in 2001. Since then, Quake II has been ported to numerous platforms and as a result, it is a good candidate for porting to the Virtex FPGAs.

To Linux or not to Linux

There were two main ways to port Quake II onto the Virtex 5FX boards: with Linux, without Linux. The advantage of porting Quake II without Linux are:

1. More control over underlying infrastructure
2. Avoid potential overhead from running Linux

The advantages of porting Quake II with Linux are:

1. Complete network support
2. Complete input support
3. Complete sound support
4. Complete graphics support

Complete support under Linux means that there should be no additional code written for the each of the modules to work, at least in theory. This is in contrast to without Linux where there is partial support in the sense that there are sample code that demonstrates working network, input, sound, and graphics support but does not completely implement the stack in a way that Quake II can readily use it. Because of the complete support under Linux, we opted to use Linux as a foundation as opposed to writing each stack ourselves.

Linux who?

For getting Linux to run on the Virtex 5FX board, we had three options: Linuxlink by Timesys, Monta Vista Linux, or a custom handcrafted Linux distribution based off of the Xilinx tree. The first two options were ideal because they provide a complete tool set for developing for embedded devices such as the Virtex 5 FX but they are also commercial distributions. Since it would be nice to be able to use existing tools, we contacted both Timesys[2] and Monta Vista for a possibility of an academic license. Timesys was generously offered us a three month academic subscription to Linuxlink, so that is what we decided to use.

Mounting from the Clouds

The first challenge that we faced with getting started with Linuxlink was the fact that it used a NFS share as the root by default. In fact, it used an NFS share specified by the DHCP server. This actually spawned a discussion on the advantage and disadvantages of using NFS share as opposed to trying to directly read off of the compact flash card. We concluded that NFS is actually the better approach because it allows the software side to be updated independently of

the hardware side. We also felt that NFS working would also be a good indication that networking is working properly.

Mapping the World

Even though we had decided to run with NFS, we still had to figure out how to specify the particular NFS root that would be mounted instead of relying on the DHCP to push one down (mainly because DHCP is served by CMU WiFi which we do not have control over). As it turns out, under the Xilinx Linux tree has support for a device tree file that maps includes booting and hardware parameters. The hardware parameters are used to tell the kernel which device exist on the FPGA (such as display and ethernet) and are directly used by the Xilinx drivers. The device tree file is compiled into the kernel altogether as a final ELF file that can be loaded onto the board directly.

The Factory

The Factory is where most of the interesting things with putting together the base distributions happen. It is a tool created by Timesys that automates the process of fetching and compiling relevant packages for the desired configuration. The Factory made it significantly easier to pick and choose required software packages and target configurations. The packages are all provided by a repository operated by Timesys. There were four main modifications we had to modify from the default Factory build. We had to replace the default Linux kernel 2.6.29 with the development branch from Xilinx to get an updated version of all the drivers that were compatible with relatively recent versions of the IP blocks available along with a custom kernel config that enabled the said drivers. A custom device tree that matched our requirements (display, keyboard/ mouse, and sound in addition to ethernet) was also used. Lastly, the factory configuration file was modified to enable hardware floating point.

The Olde World

One thing that we quickly discovered after setting up the FPGA and the Factory was that the drivers were not able to detect the hardware. After closer inspection on the drivers side, we found out that the Xilinx drivers expected and only supported older versions of the IP block than ISE 11.1 wanted to use by default. Changing the IP blocks unfortunately wasn't a matter of simply picking a different version because the IP blocks were not meant to be backward compatible (timing constraint changes was a big deal). So we had to refer to the manual to see what each IP block expected and try to figure out how to get them to coexist.

Speaking French

After getting Linux up and running on the FPGA, the next step is to get Quake II compiled and running under Linux. Since we weren't planning on compiling on the FPGA, we needed cross compiling tools to compile PowerPC binaries on our own computer. Fortunately, the Timesys Factory also generates a set of tools that allows you to cross compile binaries with relative ease. The Makefiles in Quake II still had to be modified to accommodate what was and wasn't needed (mainly to turn off OpenGL bits).

Black Screen of Penguins

For screen output, we decided to add the dvi controller to the starter code that Timesys provided, which did not work for us. After spending some time trying, we eventually decided to start with the starter code provided by Xilinx which included screen output but lacked FPU and compatible networking.

Mind-control

Adding in the PS/2 support was relatively easy. We just had to add in the PS/2 IP blocks and

enable the corresponding drivers in the Xilinx Linux kernel. The only thing was figuring out which driver to enable because there are two different PS/2 drivers. As it turns out, one of the two is a legacy driver which didn't work with Virtex 5FX and the other driver worked perfectly. The only remaining issue with keyboard/mouse input is that the Xilinx keyboard driver buffer can overflow if the user decides to hold down a key for a significant amount of time.

Racing at the Speed of Light

The soft FPU for the Virtex 5FX boards have two primary operating modes: low latency, high bandwidth. The low latency mode clocks the FPU at 133MHz where as the high bandwidth clocks at 200MHz. We opted to run with the 200MHz option because for our application, it is more beneficial to be able to run more floating point operations. After adding the soft FPU, we also had to modify the Timesys Factory to generate binaries using hard floating point. This resulted in a 25% speed up of the floating point code.

Can you hear me now?

Unfortunately, we weren't able to get sound working, although if we had more time we would have been able to get it working. The Xilinx sound driver under Linux wasn't designed the same way the other Xilinx drivers were designed. Further more, both the IP blocks and the Linux driver were originally designed for the ML405, the predecessor to our current board, which may explain some of the difficulties we were having.

Vector Coprocessor

The default 3D rendering engine for Quake II is entirely software based and built into the game itself. Versions of Quake II using OpenGL were created long after the initial release, once

the Quake II code code was open-sourced. This presents the perfect opportunity for rewriting the rendering engine to make use of a custom FPGA coprocessor module.

At it's most basic level, 3D rendering involves performing the same operation across a large set of data. When a model is rendered, all the vertices of the model must be transformed into the screen space and all the pixels within each triangle must be rasterized. Graphics cards work by operating on these sets of data in parallel. Most modern CPUs include instructions for operating on small vectors of data, usually consisting of 4 32 bit floating point numbers. These instruction set extensions go by the name of SSE on x86 and VMX or AltiVec on PowerPC. Since 4 element vectors and 4x4 matrices are used heavily in 3D graphics, these vector operations are ideal for use in graphics routines.

The FCB

Though the PowerPC core on the Virtex 5 does not natively support the execution of these vector instructions, it can decode all VMX memory operations as well as 16 user defined instructions (UDIs) and send these decoded instructions out over the Fabric Co-processor Bus (FCB) for execution. The FCB protocol is extremely low-latency, with the ability to completely process one instruction every cycle, should the coprocessor support it. The load and store operations support up to 128 quadword-aligned bits, as well as several cache-hinting and auto-incrementing instructions designed to allow large contiguous blocks of memory to be processed quickly. Each of the 16 UDIs also supports a 5 bit extended op-code, allowing each UDI to encode 32 individual operations.

A detailed description of all FCB transactions can be found in the "Embed Processor Block in Virtex-5 FPGAs" reference guide, under the "Auxiliary Processor Unit Controller" section.

The VPU

As it's designed around the PowerPC VMX instruction set extension, the FCB presents an ideal interface through which to implement a vector coprocessor. To maximize speed, we designed the coprocessor to support same-cycle response wherever possible. Since we implemented only "autonomous" instructions, which do not return data, status, or exceptions to the CPU, and a store which does not modify internal coprocessor state, the only required delay in the protocol, is that the first decoded UDI in a sequence must take at least two cycles to complete.

The single-cycle response for stores (from the coprocessor to memory) presented some timing challenges, as it requires an asynchronous register read which is not supported by the BRAMs. The lack of BRAM support is not a significant issue, as the coprocessor has only 32 registers requiring a total of 4KB of memory. In order to solve the timing issues, however, we had to ensure that very little logic was in the path leading from the control signals to the register file inputs and from the register file output to the store data line. With some tweaking of the control module, however, we were successfully able to synthesize with an FCB clock of 100MHz.

VPU Issues

Considering that the Xilinx Virtex 5 PowerPC FPU IP also operates over the FCB, this coprocessor may sound like it would obviously lead to significant speedups, especially considering that we used single-cycle implementations for all coprocessor operations except for SQRT. However, we discovered that it is possible to synthesize the Xilinx FPU with an FCB clock of 200MHz. This means that in floating point operations, only a 2x speedup is possible. In addition, since it operates directly on the PowerPC registers, the Xilinx FPU does not need to write data to RAM before operating on it.

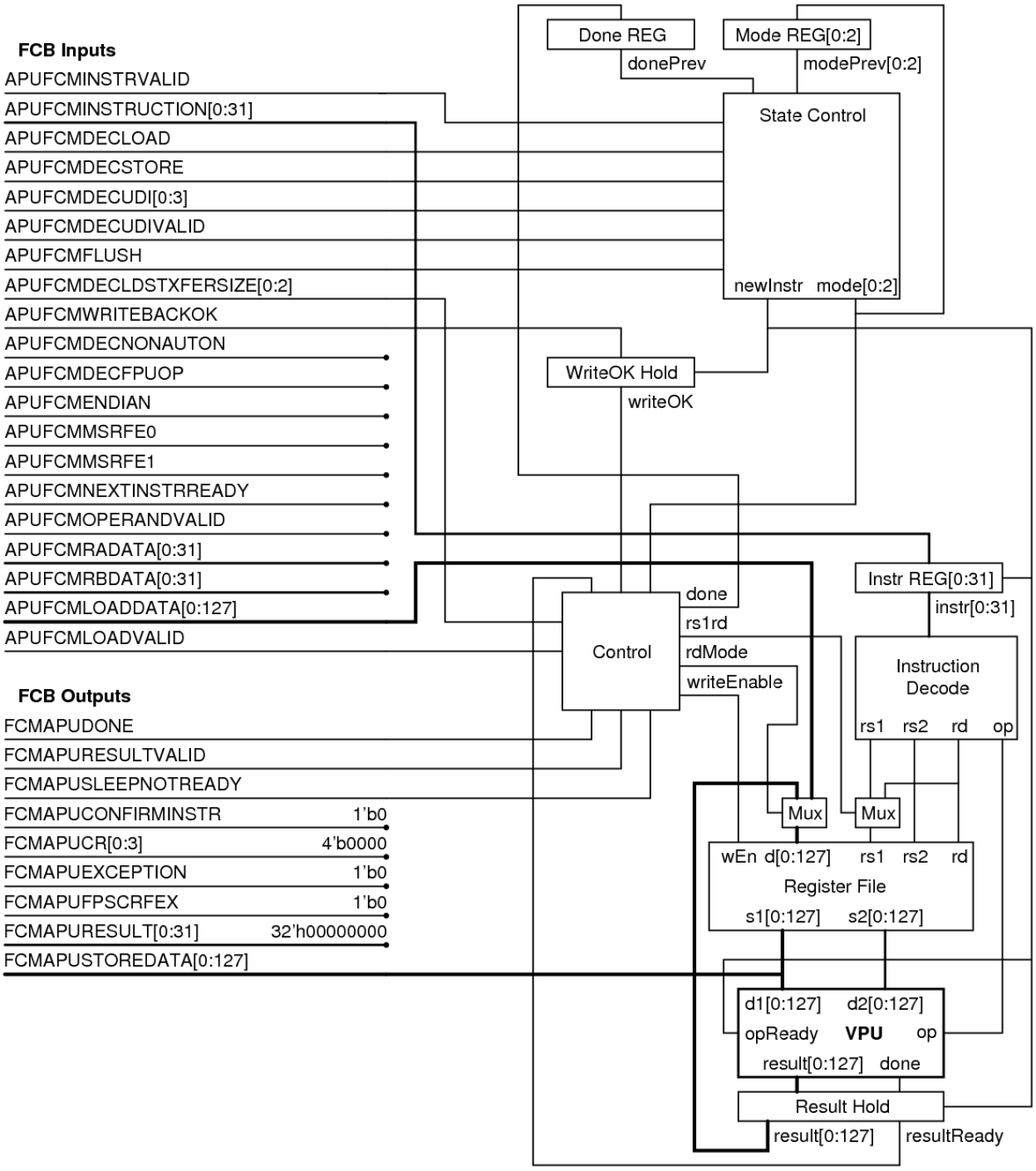
One possible solution to this bottleneck is to rewrite the Quake graphics routines in a way that vertex and pixel data does not go through the CPU registers. By avoiding the CPU registers entirely, the additional cache management offered by VMX loads and stores would theoretically

allow for more efficient data transfer than the existing CPU-based graphics routines can achieve. However, in order to implement such routines, the vector coprocessor would need to be extended beyond basic vector arithmetic operations.

XPS and the FCB

For whatever reason, the version of XPS included with Xilinx ISE 11.3 is unable to create custom FCB pcores. We were able to get around this limitation by starting with the sample code provided with Xilinx's "Expanding the PowerPC Instruction Set for Complex-Number Arithmetic"[1] and modifying the pore definition files to point to our own Verilog source files. XPS also does not include the ability to easily add ChipScope blocks attached to the FCB. (XPS fully supports the creation and debugging of PLB pcores.) In order to debug the coprocessor control module on chip, we duplicated all FCB signals as outputs from our custom pore and attached a ChipScope ILA module to these outputs. This step proved to be essential, as portions of the FCB protocol were somewhat unclear as presented in the Virtex 5 reference material. By observing the actual transactions occurring on the FCB, we were able to debug some of the corner cases which we had not implemented correctly.

VPU Control



VPU Arithmetic Implementation

The FCM that we implemented performs a relatively small set of floating-point operations

fundamental to graphics processing. However, each function is applied to a vector of 4 different values simultaneously. The register file internal to the FCM consists of 32 registers, each 128-bits wide. Each register is considered to be four individual 32-bit floating-point values.

A list of desired operation support for the vector coprocessor follow:

Instruction	Function	Expression (X={a0,a1,a2,a3})
FPAdd	performs element-wise addition across specified vectors	T = [(a0+b0), (a1+b1), (a2+b2), (a3+b3)]
FPSub	performs element-wise subtraction across specified vectors	T = [(a0-b0), (a1-b1), (a2-b2), (a3-b3)]
FPMult	performs element-wise multiplication across specified vectors	T = [(a0*b0), (a1*b1), (a2*b2), (a3*b3)]
FPDiv	performs element-wise division across specified vectors	T = [(a0/b0), (a1/b1), (a2/b2), (a3/b3)]
Mov	move specified vector to target vector	T = A
Inv	computes multiplicative inverse of specified vector's elements	T = 1/A
Sqrt	computes square-root of specified vector's elements	T = [sqrt(a0), sqrt(a1), sqrt(a2), sqrt(a3)]
Sum	computes the sum of all four elements of a specified vector	T = [(a0+a1+a2+a3), b0, b1, b2]
IToF	converts all elements of specified vector to floating-point	T = [iToF(a0), iToF(a1), iToF(a2), iToF(a3)]
FToI	converts all elements of specified vector to integers	T = [FToI(a0), FToI(a1), FToI(a2), FToI(a3)]

Within the specific implementation of the Vector Coprocessor, four floating point units provide the core of the functionality. We found an open-source floating point unit at www.opencores.org, which we were able to use for most of these functions, with the exception of division. In the case of division the open source units worked properly behaviorally, but contained the behavioral verilog abstraction "x = opa / opb;", which could not be synthesized

onto the FPGA. In place of this portion of the code, we inserted a simple combinational array division unit designed to simply provide the necessary functionality without much respect to size or speed, which did not prove to be problems in the given implementation. Aside from the division setback, the open-source FPU adequately provided addition, subtraction, multiplication, inverse (by way of simply dividing one by the operand), and conversion functionalities. A sample, small-scale diagram of the combinational fixed-point division unit that we implemented in place of the behavioral descripion is included below. A modified version was only needed to be applied to the significand in the floating-point division.

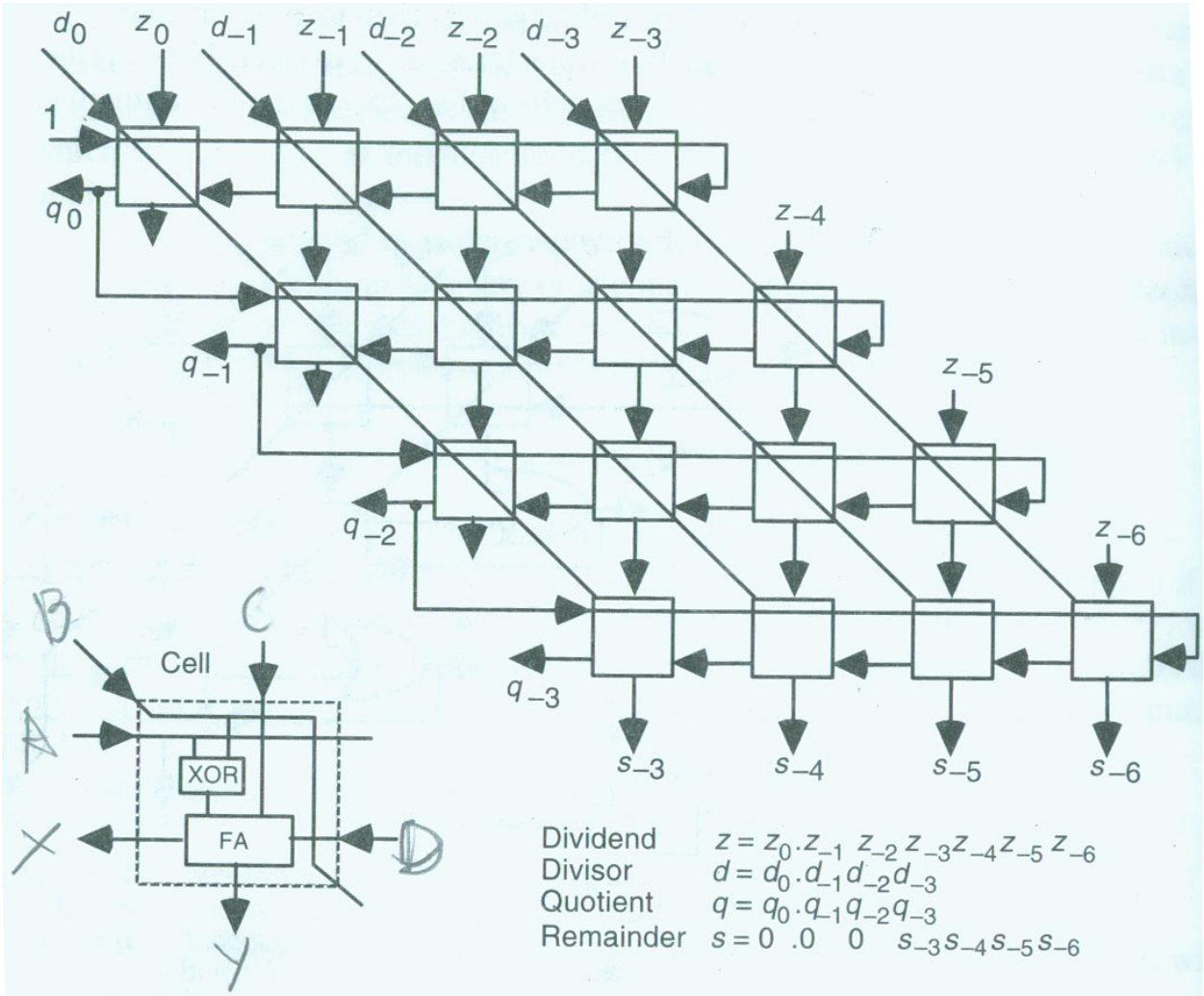
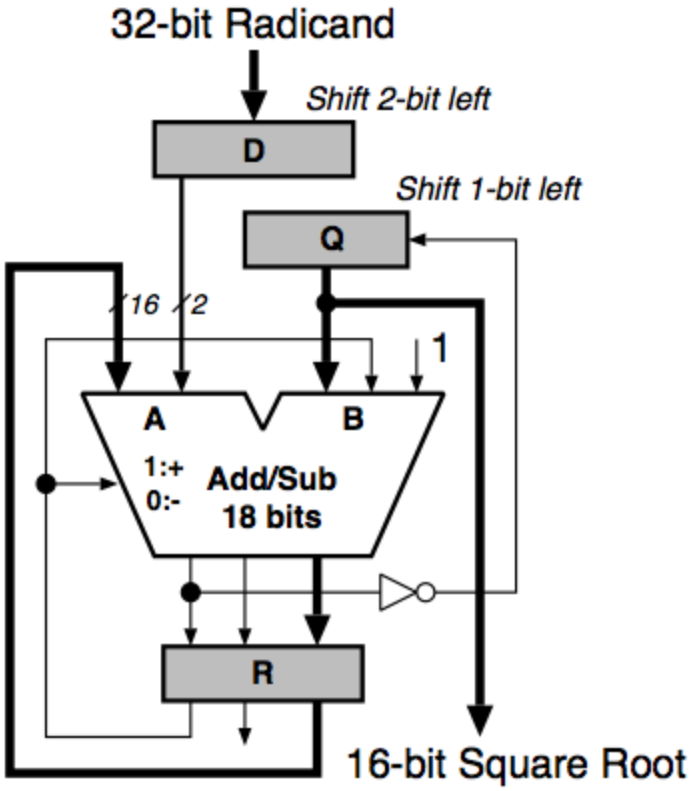


Fig. 15.8 Nonrestoring array divider built of controlled add/subtract cells.

We determined that an iterative approach was necessary in order to achieve square root functionality. We implemented and tested in verilog modules represented symbolically by the below diagram, which, with appropriate significand and bit-width sizing parameters achieves accurate square root and remainder approximations in exactly 13 perceived clock cycles. Other than the external square root module, however, most of the functionality remains within the FPUs and relies simply on input/output routing for validity.



Aaron's Notes

Over the course of the beginning of the semester I contributed somewhat in various occasional areas involving the initial lab work or Virtex 5 transition work, but the core of my contribution this semester was in the development and testing of the Vector Coprocessor functionality. For the beginning stages of our project I tried to apply myself where able, but my ability to contribute with respect to figuring out how to run linux on the FPGA were significantly limited by my lack of experience in operating system functionality relative to my partners. However, all things considered, there was a large amount of effort and time necessary as well to design and debug all of the verilog coding involved in the Vector Coprocessor arithmetic.

On the whole I greatly enjoyed the course. On the technical side I thoroughly enjoyed being able to dive into and modify an FPGA, and on the project-oriented side it was very fulfilling to be able to demo our project as a (mostly) fully-working product with clear achievements. My biggest regret in the project is that the Vector Coprocessor wasn't finished in time to really demonstrate its functionality. Moving forward
with an open-source FPU instead of writing one from scratch was a major step up in terms of schedule time, but there was still a lot of work to be done in testing that design and implementing square root, etc.

I also felt that there was a lot of time lost in several inevitable external factors. Personally I initially had a difficult time understanding exactly how to approach the Xilinx tools other than under the directions of the lab materials provided. Even without needing to transition part-way through to the Virtex 5 boards the ISE tools were somewhat challenging to get used to. Similarly, when we did actually receive the Virtex 5 boards, we essentially had to relearn some of the same techniques without the benefit of the labs due to updated IP, incompatibility issues, etc.

Alfred's Notes

Over the course of the semester, I contributed mainly to three distinct portions of the project. During the first few weeks, I worked on Labs and initial research on options for running Quake on the FPGA. During the subsequent month or so my main contribution was to look at the open source options for running Linux on the Virtex 5, and for the remainder of the semester, my most important contribution was to understand and implement all of the non-arithmetic details of the vector coprocessor and it's connection to the CPU.

In looking at open source Linux options, I spent a couple of weeks figuring out how to use Buildroot to cross-compile a PowerPC root filesystem for Linux. Though I was able to build root filesystem with some basic tools and libraries, I found that many of the options Buildroot provided did not work without much trouble. In the time I spent looking at Buildroot, I was not able to successfully compile with hardware floating point enabled, and I had a difficult time pointing Buildroot to a copy of the Linux kernel which was patched with Xilinx PowerPC support.

For the vector coprocessor, my main contribution was the control module. Though the initial implementation of this control module was simple, debugging the problems turned out to be quite time-consuming. The Virtex 5 PowerPC guide provides only basic descriptions of Fabric Co-processor Bus (FCB) transactions. In general, One waveform with a short text description is provided for each kind of transaction. The difficulty is that these waveforms assume that the PowerPC asserts all control lines as soon as allowed. They don't always make it obvious exactly which control signals could be delayed because of, for instance, a cache miss, and what the correct response to those delayed signals should be.

As an example, during a store operation, it appears that it is possible for the powerpc to delay asserting the APUFCMDECLDSTXFERSIZE[0:2] signal, and that the coprocessor should not consider the store to be complete until this signal is nonzero. In my control interface I had assumed that, since the APUFCMDECLDSTXFERSIZE is essentially decoded directly from the

instruction, waiting for the APUFCMDECSTORE and APUFCMINSTRVALID to be asserted was sufficient to decide that the store data should be presented for one cycle before moving on to the next transaction.

Though ChipScope provided an excellent way to visualize the actual transactions occurring in the FPGA, setting it up and testing new designs took quite some time. In particular, when you create a ChipScope peripheral in XPS, you generally add the signals you're interested in to a large data/trigger bus. The only problem is that when you synthesize this and connect the ChipScope software to the FPGA, you are presented with exactly that, a single large data bus. The ordering of the wires was rather non-obvious at first, with the individual trigger busses (up to 16) appearing in ascending order, but the connections in each trigger bus appearing in reverse order. This makes sense to some degree, considering that this reverse ordering appears to be the convention for PowerPC, but correctly labeling the signals in ChipScope took some time.

Even with ChipScope setup, my progress was slowed by the need to re-synthesize between attempts. As I was generally guessing at why a transaction had failed, the only way to test was to re-synthesize and watch exactly how the PowerPC interacted with my new design.

The Class

I think the largest setback for many of the groups this year was evaluating and possibly switching to the Virtex 5 boards. For us, the switch proved to be well worth it, as the Virtex 5 FXT is much better supported by Linux than the Virtex 2, but even then, the switch essentially forced us to start the major project work about one third of the way through the semester. I think perhaps we could have done more work researching Linux on the Virtex 5 before actually receiving the board, but, of course, we didn't actually know we would be receiving the board until a week or so before we received it. Overall, I think that both the decision to tell groups about the possibility of switching and our group's decision to use the Virtex 5 were good, but it certainly did certainly reduce what we were able to complete.

As others have said, I think the labs could be made more useful. When rewriting the labs for the Virtex 5, assuming this hasn't been completed already, I would suggest making the labs more like guides to using a particular feature, instead of step-by-step tutorials. By telling people where to find out how too use a particular feature and how to sort through that information, the labs would more closely simulate the process required to complete the projects themselves. At the very least, I think the labs should avoid providing starter projects. Instead, they could go over how to create the necessary project using XPS and provide some starter C code. Because of the labs, my first interaction with XPS was being prompted to update the lab's starter project to use components from the new XPS. I had absolutely no idea what many of the prompts meant. The process of creating a new project, on the other hand, appears to be one of the more pleasant thing to do in XPS, as everything should generally work first try.

Benson's Notes

I was responsible for getting Quake II to run on the FPGA. This includes putting together the hardware that fulfills our requirements (graphics, ethernet, keyboard/mouse, and sound), setting up Linux to work with the FPGA, and lastly getting Quake II to run on the said FPGA.

One thing that was definitely unexpected was the absence of a FPGA setup that did exactly what I wanted because I had originally thought that it should be a relatively common task for someone running Linux on FPGAs to want to drive video and have some form of input. Unfortunately, it wasn't so, so I spent a great deal amount of time understanding how to put various IP blocks together. Both a blessing and a curse is the thoroughness of Xilinx's documentation. Although everything we wanted to do is very well documented, so was every other possible aspect of the various IP blocks. This made digging through the documentation a very long a laborious task.

Another unexpected time consuming task was getting Linux Xilinx drivers to run. Although Timesys made running Linux easy, there was a mismatch in the IP block version between what the drivers supported and what the ISE wanted to use--namely, the ISE wanted to use IP blocks too recent for the drivers to support. In order to debug and figure out the problem, I still had to dig into a substantial amount of the kernel.

Quake II finally ran some 2AM on Thanksgiving. Then my focus shifted to getting the FPU and sound working for the rest of the time available. I was able to get the FPU working in time but unfortunately, I wasn't able to get the sound to work.

Overall I enjoyed the class very much. I definitely liked working with the FPGAs even though it was at times very frustrating, the results were definitely worth it. I think getting the Virtex 5 FX boards from the beginning of class would definitely have helped as well, because the Virtex 5 FX boards definitely set us back by two or three weeks in terms of investigating and learning how the boards worked. In fact, I wish that I had more experience with FPGAs coming into the

course instead of picking it up as the course went on. It was an unknown that proved to be extremely significant in our progress.

[1] Endric Schubert et al.: http://www.xilinx.com/publications/xcellonline/xcell_66/xc_pdf/p44-47_66_F_101.pdf

[2] LinuxLink Timesys: http://timesys.com