XilQuake

18-545 Fall 2007

Eric Buehl
Matthew Douglass-Riley
Mitchell Jewell

Our final project resulted in a fully-playable port of QUAKE by Id software on the provided Xilinx Virtex II-Pro development board from Digilent including support for networked multiplayer over a local area network. Game sound effects are mixed and rendered to the on-board AC97 audio codec. User input is achieved through the PS/2 keyboard and output graphics are rendered to the VGA controller. All maps from the original game are playable in our implementation since the game data is almost entirely unaltered. Additionally, network clients may play with other clients of any platform since the original quake protocol is also preserved.

Quake Background

The Quake engine, released under an open-source license in 1999, is freely available online. There are many ports and modifications developed by the community, but we will be focusing on the original release from iD software. This is a large code base with (over 100 KLOC) including several features we neither want nor need for this project:

- OpenGL support
- Windows support
- Optimized x86 assembly

Thankfully, the Quake engine is very modular and well-designed, making it easily modified. Most notably, all drivers (sound, network, graphics, input, and "system") exist in separate .c files exposing a common interface. There exists in the code base a "null" driver for each device, included to aid porting. These null drivers have been the launching points for our efforts.

Breakdown of Initial tasks

Task: Get engine to compile and run as-is

The quake engine comes ready to build in the Visual Studio environment under Windows. The Game runs and is playable once the appropriate game data (PAK/WAD files) have been provided.

Task: Get engine to compile and run without OpenGL

The project file for the game includes separate build targets for both hardware (OpenGL) and software renderers so this was trivial.

Task: Remove all x86 specific code

As a first implementation as well as to aid porters, C versions of all assembly functions are included. Once all of this code was replaced, we confirmed that the game still compiled and ran.

Task: Begin removing Windows-specific code

Given that we are building a target called “WinQuake” there is bound to be a lot of Windows specific code. Because of the pluggable driver system, all Windows code exists in the Windows version of each driver. All such drivers were replaced by their null counterparts, which allowed the code to compile and provided a basis for all future work.

Task: Build under GCC

Using all of the stub drivers, we were able to compile the game and watch the demos run over STDOUT.

Task: Compile with the Xilinx tool chain

We started a new project and successfully compiled all of the relevant source files for the PowerPC processor on the Xilinx board. Quake’s innate portability made this task much easier than it could have been.

Task: Run the game demo on the board

The compiled executable (ELF) file is over 1.5 MB, meaning it could not simply be marked to “Initialize BRAM.” SystemACE was pointed out to us as a potential solution and, with a nontrivial amount of work, we were able to package the FPGA configuration (bitstream) along with the executable file into a .ace file on the CompactFlash card. SystemACE’s file system operations left something to be desired (namely seeking and nonlinear access), so we added a step to the loading process wherein all game files are copied to an in-memory file system that provides greater speed and

random access. A side benefit of this approach is that the CompactFlash card is not needed after this loading completes, allowing the same CF card to be used to boot multiple boards.

Task: Display video to VGA

The game provides a video structure which, among other things, stores the number of rows, number of columns shown, number of columns total, and a character buffer. The game stores only color indices in its internal buffer, and passes this buffer along with a palette (mapping 8-bit indices to 24-bit RGB values) to the video driver. We store the palette and use it to update the entire frame every time the update function is called.

Task: Read input from keyboard

The PS2 controller only provides the last scancode value read from the keyboard, meaning Quake's standard method of polling for input between frames was insufficient. We therefore researched Xilinx' preferred method of installing interrupt handlers, and created such a handler for the PS2 interrupt. We also wrote a scancode-conversion function to move keys into the character set Quake is expecting. These keypresses are only buffered in the interrupt handler (to keep processing there to a minimum); the actions the keys are processed using Quake's existing procedures, called between frames.

Task: Profile floating point operations

We've counted both the frequency of floating point operations and the approximate number of cycles for each operation. Based on these data, we know that addition and multiplication are called much more often than any other operations. We also know that division, which we expected to require many more cycles than addition or multiplication, is actually only twice as expensive as multiplication, and only four times as expensive as addition.

Task: Mix sound and output it to the speaker

Quake includes a sound-mixer library and a generic DMA sound driver that will work with any device that uses a circular buffer. The AC'97 codec on the board uses a FIFO structure, meaning a nontrivial intermediary was necessary. In addition, the AC'97's FIFO has only 512 entries, which, at 11 kHz, means the FIFO needs to be replenished at least twenty times per second. Given that Quake only "paints" audio between frames, this presented a problem. Our solution was to take advantage of the interrupt line off of the AC'97 module that fires when the FIFO is half-empty. Our interrupt

handler replenishes the FIFO from a much larger circular buffer (currently 8192 entries) that Quake may, in turn, treat as a regular DMA device.

Further Discussion

Floating Point and Hardware Implementation

GCC supports an efficient software implementation of floating point operations. This gets silently linked in when compiled with the Xilinx tools. On the 300MHz PPC cores, we should expect to get something in the ballpark of 1MFLOP with the software emulation. This will likely not be enough since most of the graphic code relies heavily on floating point operations. However, the software floating point has consistently exceeded our expectations of a non-hardware solution.
Our attempts to move floating-point operations to the board have to this point been stymied by consistently slow bus transactions. Since any floating point operation requires at least three transactions on a 32-bit bus, and each transaction takes on the order of 30 or more CPU cycles on the PLB (and more on the OPB), neither of these busses provides acceptable performance for these speed-critical operations.

Our most recent attempts as of DR3 involved using the On-Chip Memory (OCM) bus, which, given its simplistic nature (single-master, no arbitration), shows great promise in speeding up these operations. Early tests show that we should be able to complete a full FP operation in 12 cycles. Our efforts to move FP to the OCM have been hindered, perhaps however, by spotty and inconsistent documentation that refers mostly to the Virtex-4 series rather than the Virtex-II. Eventually, all the bugs were worked out and floating point operations take place roughly ten times faster than in software.

We have considered the possibility of implementing nontrivial graphics functions in hardware; however, few if any of the methods Quake calls take one or two arguments, perform a lot of computation, and then return one or two arguments. Since argument-passing has proven to be the most complex and time-consuming aspect of moving methods to hardware, we therefore concluded that moving small, very frequently used operations (FP) to hardware remained our best bet.

The low level operations were provided by the Xilinx CoreGen utility provided in ISE. This has several benefits. First, it allows us to customize parameters of each operation for considerations such as latency and FPGA resource consumption. Each of these parameters were selected to minimize latency at the expense of FPGA area and synthesis time. An additional perk of CoreGen was the

allowance it gave us to focus on speed since we could ensure that the correctness was always preserved. Lastly, since CoreGen modules are tailored for specific FPGA resources, they are able to make explicit use of on-board features such as the 18x18 hardware multipliers present on the Virtex II.

Booting/ELF loading

With the initial help of the TAs, we determined that booting arbitrary sized images should have been possible using the onboard SystemACE controller. The documentation for SystemACE describes it as an all-encompassing solution for loading a variety of runtime-selectable options including software, board configuration and FPGA bitstreams. However, this did not work as promised in our first attempts. The solution to this problem countered the original statement in the lab 1 writeup which said, “You probably won't need on-chip debugging, so disable it for now.” Since SystemACE relies on the JTAG interface to set up the initial state of the PowerPC core (among other tasks), this was causing it to fail. Once a JTAG IP core was synthesized, SystemACE loading worked. Since SystemACE takes care of core setup and binary loading, this removes our need to rewrite these parts.

Audio

The original Quake game used small wave data (stored in the PAK files with other game data) for in-game sound effects. Game music was stored as CD audio on the the game CD which obviously required that the CD be inserted during gameplay. Ideally, we would like to encode this music into MP3 or another compressed format and store it on the CF card. Either a hardware or software implementation to decode these data will be needed. Specific hardware decoders exist that are relatively simple to interface with. If the resources are available, a software decoding routine will likely be the simplest to implement as there are several integer versions of audio codecs. In either case, game music will still need to be mixed with the other in-game effects, so if decoding is done on auxiliary hardware, it will need to be fed back to the FPGA for mixing.

Quake supports 8- or 16-bit, mono or stereo audio. We have chosen 16-bit stereo audio since it matches most closely the particular format expected by the AC’97 codec.

Networking

Xilinx provides two networking libraries: Xilnet and, more recently, lwIP (“lightweight” IP). The latter is an open-source library that supports TCP and UDP along with a number of other potentially useful IP protocols. For example, lwIP supports DHCP, which could be used to provide IP addresses to several boards in anticipation of a several-board multiplayer match. lwIP only requires

that a data-link (layer 2) driver be provided to enable its functionality; such a driver for Xilinx' Ethernet IP is already available.

Since the interrupt structure was already in place, and Quake's modular driver architecture simplified this process, the final implementation of networking took only a few days. Then we performed a lot of tedious first-hand testing.

Personal Notes for Eric

The majority of my time spent on this project revolved around performance. At the core of this was the various implementations of floating point hardware. Before we began any actual implementation, most of my work could be approximated as research. I began by compiling a stripped-down version of the game engine under linux on an x86 workstation. This proved to be extremely useful later on as well. The first major challenge in this endeavor was correctly linking against the libc software floating point emulation routines. In order to not need the true software implementation (since most libc distributions do not include this support), I created a series of wrapper functions which would compile into true hardware operations. The advantage of this was the ability we now had to profile at the granularity of single floating point operations as opposed to boarders on function calls. It did not give an accurate measure of time, but it gave us an approximate count of each operation. Also, it exposed the best way for which we could patch the code compiled from GCC to use our future custom hardware. We simply implemented the relevant softfloat functions to interface with our hardware and GCC took care of linking. Initial profiling consumed about the first third of my time on this project.

FPU

While the actual implementation of the floating point operations was mostly provided through CoreGen, there was still a need for support infrastructure. The first incarnation of our floating point hardware operated on the OPB bus. With a single operation, the latency was so high that this made no measurable increase in speed. In fact, one of the first implementations actually slowed down the game framerate. From here, I quickly transitioned our hardware to the PLB bus but found that this was no better. We now had two options – using the DCR bus or the OCM bus. Both were fast and simple, yet neither were well documented. After several stumbling blocks, I was finally able to get a reliable floating point point core operating on the OCM bus that played nicely with all other hardware and software routines. This portion was ongoing throughout the project and constituted the remaining time spent.

Speedups

In the final weeks of the project, I had also discovered the ability of the the processor cores to run 30% faster (up to 400MHz). This ended up being a relatively minor change, but added a significant

performance boost. Coupled with the floating point hardware, our framerate was increased several times that of the original pure-software version. I was also able to get on-board profiling working in the last week of the project, however, this did not prove useful in the time remaining.

Hindsight

My first vision of this class was that it was something I would not enjoy. Rather, I was delightfully surprised that it was exactly something that I wanted to do. A big part of this came from our choice in projects. I had no particular interest in the actual implementation of a game from scratch and I was glad that the rest of my group agreed with this mentality. Another perk was our "group dynamics". Both Mitch and Matt's backgrounds are in software and since much of our work involved software, they were right at home. Consequently, I was able to focus on more low-level tasks.

We were very lucky in that our projected paralleled extremely well. Especially once we got a second workstation, we could work much more efficiently. Also, being able to use remote desktop on the lab machines saved lots of time by not having to necessarily be physically in lab.

One thing that Matt and I discussed several times was the over-all disdain for Xilinx and how the name "Xilinx" became the scapegoat of every problem. While it is true that there are *some* deficiencies in their software, it is not the swiss cheese that the course staff makes it out to be. There were several instances of "bugs" which I would automatically attribute to some sort of broken Xilinx code but later found that it was, instead, a misconfiguration that was illuminated by a different piece of documentation. This brings about another point. The Xilinx documentation, as a whole, is good. However, it is very sporadic and not well organized. I have found that no Xilinx system is described by a single "Rosetta stone" PDF. Instead, it involves a lot of searching to find the answers.

The best part about the class as a whole, was the relaxed atmosphere and total focus on the projects. As it became clear that the labs were only there to server a functional purpose, that burden was released leaving more time to focus on the project itself. Since the labs were also useful as a reference, I think it might be more useful to have more of them, yet not necessarily required. Perhaps if they were viewed more as a series of tutorials for which people could reference.

Personal Notes for Mitch

I began by working on the video driver once SysACE and the memory file system started working. I went through the Quake source code to find that it provides us with a palette of 256 rgb values and a video buffer which stores indexes into this palette. When Quake changes the palette, I copy the new palette's contents into our saved palette. Between every frame, VID_Update is called, in which I translate the one byte index from the video buffer into a four byte rgb value. I then place this value in the appropriate spot in the Xilinx frame buffer. Later, Matt added the ability to account for the 250 pixel offset and centering the picture if it is not full screen. This part of the project took me approximately 15 hours.

Next, I worked on the keyboard driver. Matt set Xilinx to fire an interrupt whenever there were data on the PS2 controller and call a handler function when this happens. In that function, I add the value on the controller to a rotating buffer of scancodes. Between every frame, Quake calls Sys_SendKeyEvents, in which I remove each scancode from the queue, decode it, and call the Quake function Key_Event on it. If the scancode is 0xF0, I send the next decoded scancode to Key_Events and tell it that the key was released. If the scancode is anything else, I call Key_Events for the decoded scancode and tell it that the key is being pressed. In order to save time, I store which keys are pressed every frame so that holding down a key does not result in a large number of identical Key_Events calls. I also wrote the table that translates scancodes to keys. This part of the project took approximately 15 hours.

After input and video were finished, I wasted a good deal of time playing Quake. I still managed to find time to work on sound though. Unfortunately, despite spending a great deal of time debugging my and Matt's audio code and searching through Quake and Xilinx code for audio specific functionality, very little of the final audio code is mine. Quake provides functions to read sounds from memory, mix sounds, and write them to a DMA buffer. I wrote the simple initialize function, and I designed the SNDDMA_GetDMAPos[ition] function, although I did not actually write it. I gave some input to Matt's writing of the handler, and debugged it at times, but the majority of my time here was spent trying to understand what the code wants and what was wrong with our code [turns out it wasn't ours that was a problem, it was Quake's]. Despite not providing much of the final code for audio, I still spent approximately 30 hours on it.

Finally, my last task was the network driver, specifically the UDP driver. Again, Matt set up the interrupt to fire when a packet is received. Quake uses a proprietary sockets protocol, built on top of an

UDP or TCP driver. We chose the UDP driver, and using lwIP (Lightweight IP) UDP functions, I implemented all of the UDP driver functions, many of which seem more like sockets functions. I also set up the function that is called whenever a packet arrives, using another circular buffer to save the packets being received. I spent around 25 hours coding and debugging this part.

As for my impression of the class, I greatly enjoyed it. By the end of the semester, it was my one class for which I wanted to do work. While I believe I learned a lot from this project, especially reading other people's poorly commented code, I would have preferred a project where we would have to create most of the code ourselves, especially programming the FPGA. I obviously do not hold this against the class, because my group partners and I agreed on this project. Retrospectively, I would have preferred to spend more time programming and debugging and less time researching.

Personal Notes for Matt

Contributions

By and large, most of my (successful) work was done in game infrastructure.

SystemACE

The first problem I tackled was getting our executable to run on the board. Compiling to PowerPC was no issue, but since our codebase is very large, the resulting executable was far too large to run from block RAM. We needed a method of loading the ELF into main memory and setting up the processor to run it. Originally, we thought we might have to write our own custom boot loader and put it in block RAM. Thankfully, that proved not to be the case.

Some research showed that SystemACE, Xilinx' technology for specifying a full "start configuration" for the board (including hardware bitstream and loaded executable code, was available and would suit our needs very well. This left the problem of getting it working. It took on the order of eight hours in lab to figure out how to create a system.ace file properly targeted to our board, and, in turn, to prepare the board to receive it and use it correctly. The general lack of experience with this technology (among students and TA's alike), along with a few very costly "cost-cutting" assumptions I took from the first lab, made this a difficult task. However, once I took the time to read through the Xilinx documentation and experiment, I eventually came on the solution: add the JTAG IP to the design such that the SystemACE controller could communicate with and initialize the PowerPC.

Memory File System

After the executable was running on the board, it became obvious that the file system capabilities provided by the "Standalone" BSP (Board Support Package) would not be sufficient for Quake's needs. Specifically, Quake required random access to files, whereas the XilFAT driver for the CompactFlash file system only supported sequential access (so far as we could see). Xilinx' Memory File System (MFS) library provided functions that would allow random access, but required that we copy files from the CF card into main memory before running the game. This proved an acceptable tradeoff, and, in fact, provided an added benefit: we are able to remove the media after the copy stage, meaning multiple boards can be booted off of the same card.

I wrote the code to create the in-memory file system and populate it with files from the CF card. I added a progress bar to make loading time more bearable, and a measure of transfer speed to

determine effects of moving the MFS around in memory and enabling caching. All told, ten hours of code analysis, coding, and testing probably went into this task.

Interrupt Fabric

When it came time to write the input driver, and we realized that once-per-frame sampling would not make for a playable game, two options were presented us: find a tight inner loop in which to sample input (the option favored by the TA's), or install an interrupt handler (and an interrupt controller in hardware to drive it). I chose the latter, since it was by far the more elegant, maintainable, and expandable option.

Eight hours in lab later, I had added the Interrupt Controller (INTC) IP to our board design, connected its interrupt line to the PowerPC core, connected the PS/2 interrupt lines to the INTC, and was receiving keystrokes mid-frame. I spent most of that time searching through Xilinx' documentation finding best practices for using the XIntc driver to install handlers. I was sincerely impressed that I didn't have to write a single line of assembly to install that or other handlers in the system—Xilinx' driver was very thorough, providing everything down to the assembly stubs. Once this was done, Mitch wrote code to queue incoming keystrokes, translate the scancodes, and invoke the proper functions in Quake to register the key state changes.

This investment proved to be a wise one—once the interrupt fabric was in place, we were able to expand it very easily for sound and networking.

Sound

Mitch was primarily responsible for getting the Quake DMA sound mixing code into our source tree. However, the AC'97 Codec on the board uses a FIFO rather than DMA, so a nontrivial mediator was necessary. Simply populating a buffer and flushing it every frame wouldn't work, since, even at 11 kHz (our audio output frequency), the FIFO would empty more quickly than frames are drawn.

To solve this problem, I connected the AC'97 to the interrupt controller, and trapped the interrupt that the codec fired when its FIFO is half-empty. The interrupt handler draws from a large circulating buffer, advancing the read pointer in the process. Quake, in turn, writes mixed audio to this buffer a sufficient distance ahead of the read pointer, and the sound plays without skipping. This was a relatively simple coding task; all told, I spent on the order of five hours on it.

Networking

Again, the investment in an interrupt handling framework paid off when it came time to provide

networking to the game. Also, again, I wrote the lower-level delivery code (the "level 0" driver) and Mitch wrote the higher-level service code (the "level 1" driver)—in this case, I wrote enough code to communicate with the Ethernet port and to properly initialize and poll the lwIP library. This was nontrivial because the right polling calls had to be inserted in the right places in the application code—even though the process of moving frames from the Ethernet IP to an in-memory buffer was interrupt-driven, the process of moving frames from the buffer to the application is done via polling.

I got the board to the point where it was responding to PING's; from there, Mitch provided UDP functionality to Quake through lwIP's own UDP interface. My part, from resynthesizing with the Ethernet interrupt enabled to the point where I could ping the board, took on the order of ten hours.

Reflections on the Course

Labs

I get the impression that the labs are panned by every year's class, and I believe they are improving. However, in our case in particular, the labs this year almost did more harm than good. Making sure that we learn about the hardware early is a noble goal; however, to teach us about it by giving us the "90% case"—really, "here's how the TA's understand the material"—is a huge mistake. In our case, the nuances of the uncommon case proved much, much more important.

I would propose that the labs be much more self-guided, providing a concrete end goal and links to the relevant Xilinx documentation, and nothing more. Don't distill the documentation. Most especially, avoid step-by-step tutorials—they're not labs, and they're not worth anyone's time.

Problems with Xilinx

Despite all our complaints over the course of the year, by and large, the platform, code, and utilities provided by Xilinx were remarkably robust and useful. I believe the course staff set the wrong tone from the beginning by giving us the impression that Xilinx is not to be trusted. That sort of thinking leads to the following problem: I have code I think should work, but it doesn't, so I blame Xilinx rather than myself. This line of thinking was very popular among the TA's, and it often proved to be incorrect.

The key insight I would hope to impress on everyone involved is: Xilinx' code is being used by thousands of people; your code is being used by about three. It's always possible Xilinx is to blame, in the same way it's possible an application is crashing because of a bug in the Linux kernel. Assume it's your bug unless you have compelling evidence to the contrary.

In totality

I'd prefer not to end on a sour note because, overall, I really enjoyed this course. I was very proud of my team's effort and our ability to describe and detail a feasible idea, work toward it and bring it to fruition. It was also amazing watching four people play Quake in multiplayer using three boards running our software. At its core, this course is about systems integration and project planning, both areas in which I was glad to get more experience. It was a sometimes frustrating, sometimes tiring, but often rewarding semester, and I know I'm better for it.