ECE532 Digital Hardware: Individual Report
Hardware Ogg Vorbis Decoder Project
Jason Zebchuk

Jason Zebchuk
990836315

April 16, 2005

1 Introduction

1.1 Initial Objectives

The goals of this project were to build an Ogg Vorbis hardware decoder. Preferably this coder was to use Lesleys grad work as a way of communicating between the different modules.

The Ogg Vorbis audio decoding procedure is actually two separate procedures. First the Ogg framing system is used to separate packets which are then decoded into audio according to the Vorbis specification. The initial plan was to do the Ogg portion in software while translating the Vorbis portion into hardware.

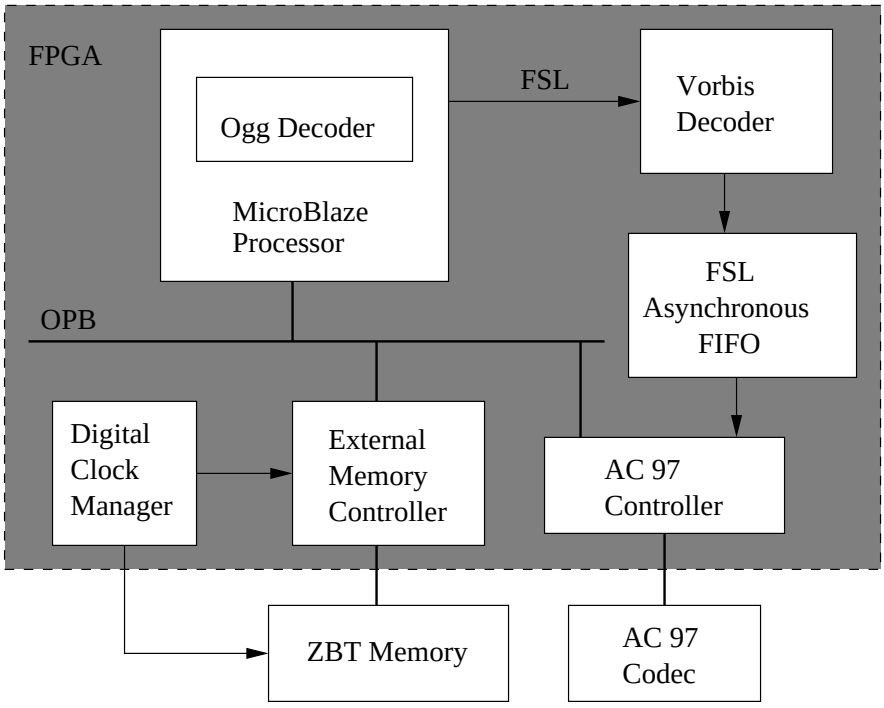

Figure 1: Initial Block Diagram

The design was to use pre-made components for the parts not directly related to the Ogg Vorbis decoding. So we used the digital clock manager (DCM), external memory controller (EMC) and FSL links from the EDK libraries. The AC97 controller used was the result of one of the 2004 classes

projects. The only components original to this project were to be the Vorbis decoder in hardware and the Ogg decoder in software. Figure 1.1 shows s block diagram of the system we initial intended to create.

1.2 Outcome

Moving the entire Vorbis decode stage into hardware proved too ambitious a goal given the time available. In the end only the overlap and add block was moved into hardware and the remaining blocks remained in software, as show in Figure 1.2. For more details on the various blocks shown in this figure, please refer to the group report on the **Hardware Based Ogg Vorbis Decoder**, by Mark Teper and Jason Zebchuk. Much work remains on this project for the future. The resulting implementation is too slow to be feasible for good quality music.

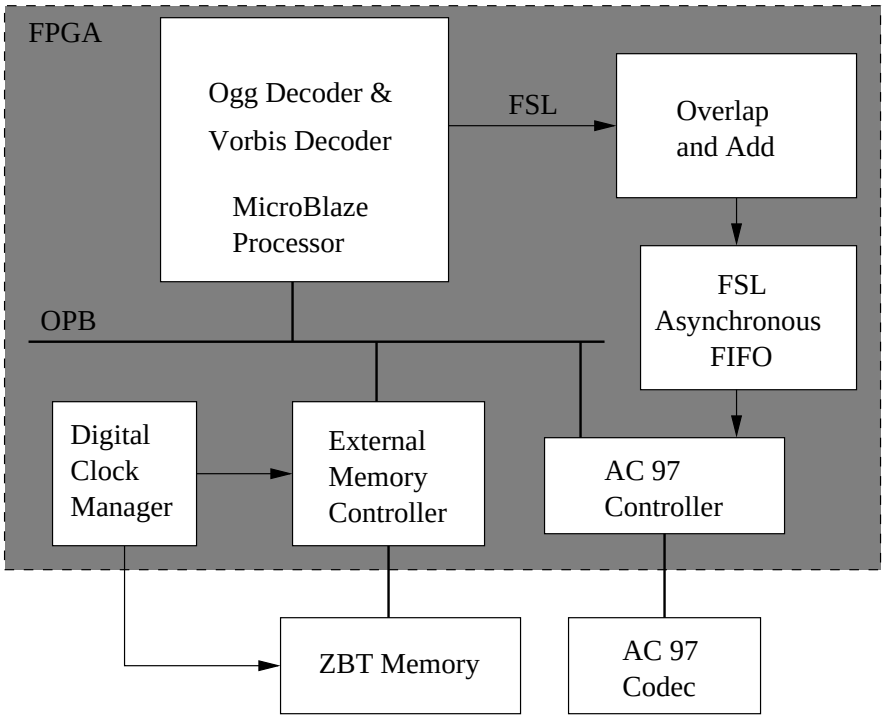

Figure 2: Final Block Diagram

To accelerate the decoding procedure, moving the inverse modified discrete cosine transform (iMDCT) into hardware should be sufficient to pro-

duce acceptable performance. This might be difficult however because the algorithm used for the iMDCT is difficult to execute in parallel. The algorithm used in the software to compute the iMDCT runs in eight different stages. Each stage requires access to the entire audio packet before it can begin, making it difficult to simply pipeline the stages. We came up with four ideas of how the iMDCT could be moved into hardware efficiently.

The first of these is to connect each phase of the transform with an FSL reading and writing the packets from the FSL into an internal BRAM and then writing them out on the next FSL. This solution would make testing easier since each stage could be done independently. However, it is fairly inefficient since it requires considerable overhead for the moving of large quantities of data through the FSLs.

The second idea is to assign a BRAM block for each stage of the algorithm. Once all the stages were finished, they would connect to the BRAM that was used in the previous stage. Thus, instead of moving the data between stages, we would just move the stages to the different BRAMS. This has the advantage of requiring less complication and overhead from transferring this considerable amount of data.

Another possibility is to select only portions of the algorithm to implement in hardware. As an example, a small part of the butterfly algorithm in the iMDCT might be implemented in hardware. Such simple transformations that perform simple manipulations on a range of data might be efficiently incorporated, but we feel this would be most effective if the hardware blocks had independent access to the memory, so that only an initial address needs to be given to the hardware block.

Finally, it may be possible to find a more efficient algorithm that is more suited to a hardware implementation since the algorithm used was designed to be an efficiently implemented in software software in mind.

2 Individual Contributions

2.1 Overview

We broke the project down into two different streams, so that much of the early work could be done individually, before the two stream naturally merged. I worked on the software stream, while my colleague Mark worked on the hardware stream.

The development of the software proceeded in several stages. The first, and largest stage was to create a simple, bare-bones implementation of an Ogg Vorbis decoder that would run on my GNU/Linux laptop. While I was doing that, Mark was working on combining the various öff-the-shelfẗhardware cores that we would need to use, to produce our base hardware platform. At that point, the hardware and software streams would merge as we ported the software to work on top of the Microblaze/FPGA platform. Then we would proceed to move software stages into hardware one at a time.

That was the original plan, and that's mostly how things happened, with a few variations as we neared the end of the term. In the end, Mark started work on the overlap and add module before I had a working version of the software, and he also spent some time on adding fixed point arithmetic to the software before I had ported the software to work on the Microblaze system. In the end, I think the two major problems that we faced were being overly ambitious to start with, and also taking too much time before beginning work on our original hardware cores.

2.2 The Software Development Process

The Ogg Vorbis formats are both open and freely available. There are also open-source libraries that are readily available to support these formats. Despite these freely available libraries, and several applications that make use of these libraries to play Ogg Vorbis file, we decide to write our own Ogg Vorbis decoder program.

The main reason for this is that we felt the time it would take to port the available libraries and an application to work on the Microblaze would be similar to the amount of time it would take us to write our own program. As well, having our own program would make it easier to move some of the functionality into hardware blocks, as well as making it easier to test that our hardware blocks worked correctly.

In addition to this motivation, I felt confident that I was capable of writing the software in a relatively short period of time. I have experience in programing in C, not only for school assignments, but also during the year I spent doing an internship as a software developer. As well, this past summer I worked as a software developer writing sound processing software in C++. I have experience in implementing a fast Fourier transform algorithm, and I felt confident in my ability to complete the Ogg Vorbis decoder. In the end, I was successful in writing an Ogg Vorbis decoder, but it required about twice

as long as we had originally intended.

The major difficulties that I faced during the development of the software was the vagueness of certain parts of the specification documents. Much of the specification is provided in the form of pseudo-code. There is some discussion of the underlying concepts, but many of the key points are described only in pseudo code. In addition, after looking at the official libraries that implement the specification, I firmly believe that at least one portion of the specification document is incorrect. Fortunately, I had access to the source of the official libraries so I could look at their implementation for guidance.

After I had implemented the specification as I understood it from the available documentation, I began trying to debug and verify my program. Using the available libraries, and an example program that came with them, I was able to verify each stage of the decode process in order. Without this, it would have taken me considerably longer to get the program working properly.

After I had finished the initial program that ran on my GNU/Linux laptop, I started porting it to work on the Microblaze. At the same time, my partner Mark began converting my code to use fixed point numbers instead of floating point. Since I had written the entire program, it was relatively easy to get it working on the Microblaze. Of course, the initial port to the Microblaze was several orders of magnitude too slow to be of any practical use. In an effort to improve performance, I changed the iMDCT implementation to use static lookup tables to compute trigonometric functions instead of computing them as required. Finally, I re-wrote the final stage of the program to use the overlap and add module that my partner created.

Almost all of my time on this project was spent working on software development. The time I spent working with the hardware, I was mostly just trying to get the software to run on the Microblaze instead of my laptop. Once Mark finished the overlap and add module, I worked on testing it on the hardware, and wrote a program to pass reference data into the module and compare the results. Thus, I was able to help Mark identify problems in his module, that he was later able to observe in simulations as to fix.

2.3 Lessons Learned

Having our own software really simplified testing and debugging. Not only was it much easier to get the code working on the Microblaze, but it was easier to remove the portion that was eventually implemented in hardware.

Even thought it took longer than expected to write the software, after looking at the code in the libraries that we might have used, I am certain that we chose the right path. Even while I was trying to verify my implementation by comparing the output of each stage with the same output in the reference implementation, it was difficult to be sure that I was looking at the correct point in the library.

As well, I was able to take the output from before and after the overlap and add stage, and give them to Mark in a format that made it easy for him to generate a ModelSim simulation to verify the design of his module. I then used the same data, in a different format, to run a similar verification on the actual hardware. Having real sample data to pass into the overlap and add module and compare with the correct results made it much easier to test the hardware module.

In addition to these benefits of having our own code, we also benefited from using C++ to write the program. Mark converted the code to use fixed point numbers, but instead of having to change every use of floating point numbers, he simply had to implement a fixed point class which overloaded the necessary arithmetic operators, then change the floating point definitions to use the fixed point class. Having object oriented code also made it easy to use the overlap and add hardware module that Mark created. I simply had to replace the overlap and add class I had already written with a similar class that instead passed the data to FSL connected to the overlap and add module. Thus, using C++ turned out to have a number of unexpected benefits.

The major problem that we faced with this project was the lack of sufficient time to implement and debug hardware modules. Our original plan was to complete the software first, then start the hardware. The main idea was that writing the software first would make it easier to integrate the hardware modules with the software, as well as providing a better understanding of the decoding process. While both of these points proved to be true in the end, we simply did not have enough time to develop hardware. It would have been much better to start designing and implementing the hardware while the software was still being written. The importance of hardware-software co-design was definitely made clear.

3 Community Contribution

At the beginning of our project, Lesley Shannon approached us with a request that we use communication modules that she was developing as part of her research. In return, she offered us additional help to incorporate her modules into our project. In the end, we decided not to use her modules because of time constraints, but I would like to acknowledge her intentions, even though we did not use her modules.

I do not think that anything I learnt would make a significant contribution to the community. However, I do wish that I had known in advance that the support for using C++ was adequate. Initially I had reservations about using C++, and I was wary of using any advanced features of the language. In retrospect, I believe my worries were undeserved, and I would encourage people to take advantage of the benefits of using C++, as mentioned above.

One of the Xilinx tools that I used that I believe few other groups were aware of is the XPS SDK. The SDK is a modified version of the open source Eclipse project initiated by IBM. Xilinx has modified the platform to work with their tools, and added an option to associate a software project in Eclipse with an XPS project. I think the use of this tool is somewhat under represented, and other groups might benefit from knowing of its existence.

4 Feedback to Xilinx

On the whole, I think Xilinx is doing a decent job with their development tools. However, there are many instances where things seemed to be *almost* right.

First, let me begin by saying that used the Linux version of the Xilinx tools for much of this project. The biggest complain that I have is that little effort seems to have gone into porting the code to work on Linux. On several occasions, I would get an error when trying to start XPS. The error was related to the middle-ware software that Xilinx used to allow their Windows based code to run natively on Linux. I was unable to find any mention of the errors I was getting anywhere on the Xilinx website, and I was only able to solve the problem after finding mention of it on a support website for a different application which made of the same middle-ware toolkit.

Another complaint that I have is the integration of the GUI and the command line tools. All of the real work of the GUI is done using the command

line tools, and as a result, I was able to do much of my development without using the GUI. At the same time, it seemed that the GUI was necessary for certain parts of the development process. The impression that I get is that one should use either only the GUI, or only the command line tools, because in using them together there is a certain lack of control that I find frustrating. If the EDK GUI was better, then it might be possible to rely on that program, but the environment it provides still seems clunky and awkward, despite the best efforts of Xilinx.

In addition to the general feedback above, here are a number of specific complaints and suggestions in regards to the Xilinx tools:

- The Linux GUIs are slow, and problems specific to the Linux applications are not well documented.
- The XPS SDK is not very well integrated with the XPS. The XPS should be more aware of SDK projects, and it should be capable of building them without running the SDK.
- More options should be added to the SDK that are specific to the XPS EDK. Specifically, it should be easier to specify things like stack size and location.
- I had difficulty connecting the software debugger to the XMD when I ran the XMD from the command line, but it seemed to work fine when I started both from the within XPS.
- It would be nice if I could use the normal gdb debugger, without the GUI, to debug software.
- The mb-gcc and mb-g++ programs seem to be based on older versions of gcc and g++ respectively. The newer versions of these programs have many advantages over older ones, especially g++, and I would like to seem updated versions of mb-gcc and mb-g++ based on more recent versions of their regular counterparts.
- In discussions with my partner about FSLs vs. customs instructions, we felt that the FSL concept might possibly have been implemented as a custom instruction. We found many uses were custum instructions might have improved the performance of our application, but where there might have been little, if any, benefit from using a module connected through an FSL.

5 Course Feedback

I appreciated the format of the labs and of the course overall. The biggest problem I found was learning how to use the tools while at the same time trying to use them as part of the course project. I think there is enough to learn in the lab to justify a full year lab. It would be nice if there was a course in first semester where we had an opportunity to learn how to use the tools before starting our projects. This would be especially helpful in giving us an idea of what we should try to do for the projects.

The organization of the lectures does leave something to be desired. The relative lack of reference materials was similarly disappointing. As a result of not having a textbook, the lectures were the only real source of information available, and with only two lectures a week it seemed that we were only able to cover a limited range of topics with a limited amount of depth. I think that the idea of creating a new text to meet the needs of this course should be considered. Or at least the possibility of providing a compendium of selected chapters from various texts to provide more depth of information of the topics discussed in lecture.

Overall, I enjoyed the course, and I think the format works well. It might be possible to start the projects a little earlier, but it would be difficult to start thinking about the project before having a chance to gain some appreciation for the Xilinx tools used during the project. I appreciate having the project end a few weeks before the end of term. The reduced workload during the last two weeks was greatly appreciated.