ECE532: Digital System Design

Audio Visualization Device

Individual Report

Jennifer Mo
(992476762)

April 2, 2007

Table of Contents

1. Introduction

The goal of this project is to create an audio visualization device that is similar to those commonly found in audio players such as WinAmp and Windows Media Player. Our final product is displayed output on a VGA monitor with 24-bit colour at a resolution of 640x480.

I was responsible for the rendering module of this project as shown in Figure 1 of the Group Report. The design methodology that my team took and the problems that I encountered during the course of this project are documented in this report.

2. Project Partitioning

The project workload was divided up into blocks according to the system block diagram found in Figure 1 of the group report. These well-defined blocks made it fairly easy for our parts to interoperate. Furthermore, the amount of system-level design that we did together before coding meant that we all had a very clear idea of what was required of each module.

Most of the interaction between our modules is through FIFOs and buses such as the OPB and FSL. The use of these well-defined interfaces simplified our integration efforts immensely. It was helpful that there was never any uncertainty on how a given interface functioned because these were already well documented. When connecting up our components, we only needed to make sure that we all understood what format the data was expected to be in.

In fact, we only had one custom interface – the ZBT RAM memory interface used by the render module to write to the frame buffers. Even though Jason wrote this interface, I had no trouble in using it. This is in large part thanks to the simulation testbench created by Jason. In addition to testing for functional correctness, I also discovered that the simulation waveforms are a clear and concise way of demonstrating how signals are to be asserted in order to use a given interface.

My primary contribution to this project was the render module. More details will be provided in Section 4 of this report.

3. Design Flow and Methodology

A major factor of our success in working together has been the Subversion version control system used by our group for source code control. This tool allowed us to efficiently review changes made by other group members.

The last point mentioned above is especially crucial to the way we divided up work in this project. Often, we only had one person adding features to the project at a time. We found that passing the multimedia board amongst the group members allowed each member to test their code immediately after simulating their newest feature. If any problems were found, they could be fixed with minimal turn-around time.

By adding only a few features at a time, we were able to ensure that we would have no integration pains later. Whenever it was my turn with the multimedia board, the source code control system allowed me to review the most recent changes. This review process allowed us to be well-versed in each other's code. It also meant that no time was wasted trying to debug problems caused by misunderstandings of how each other's modules worked.

The last step I performed before adding new code was to recompile my version of the code base and test it to see if all the newest files were checked in. From experience, this last step can save a lot of anguish caused when other group members forget to check in some crucial part of their code.

When it comes to adding a new feature, I would first design the hardware by drawing the appropriate datapath and the corresponding control logic's state diagram. After implementation, I would perform some rudimentary test cases using do scripts in ModelSim. I typically would not resort to the more involving test benches unless there are complex interfaces to consider. One place where I found a test bench to be helpful is when simulating writes and reads to memory. It was much easier to use the behavioural ZBT RAM model rather than try to assert the signals manually in a do script.

After simulation, I would test the hardware on the actual FPGA. If the test were unsuccessful, I would look at my code again and see how the observed behaviour could be produced by the code. When I think that I have found the problem, I would verify that a certain set of simulation input vectors would trigger the unwanted behaviour. Only then will I actually fix the code and re-simulate it to check that the problem has been fixed. This process will be iterated until the code works as expected when programmed on the FPGA.

4. Project Contributions

The following sections describe the contributions that I have made to this project.

4.1 Rendering module pcore

The rendering module was my main responsibility for this project. Without this module, the Microblaze processor would need to individually set each pixel when it wishes to clear and draw to the back buffer. Setting each 32-bit pixel requires the processor to perform a write to the memory through the OPB bus. The reliance on the OPB bus

severely limits the memory bandwidth available to the processor and becomes a performance bottleneck.

The rendering module alleviates this problem by hardware accelerating the rendering process. I have designed this module to be divided into four separate sub-modules, each designed for a specific task. There is the top-level rendering module, the clear_buffer module, the draw_fft_bin module, and the fade_buffer module. This subdivision of the rendering module allowed me to independently test each new operation as support for it was added to the rendering module.

The top-level rendering module contains the FSL interface through which the Microblaze processor would write its commands. This module would then read from the FSL, decode the requested operation, and then tell the corresponding sub-module to perform the operation. The FSL makes the control module exceedingly elegant. Had an OPB interface been used instead, we would have needed to store the incoming commands into our own FIFO. We would also need to worry about the command FIFO filling up. With the FSL, the FIFO buffering is done for us, and if we use the blocking FSL write, the processor will simply block until there is space in the FSL for its command.

Part of the reason why I chose to isolate the control module from the sub-modules was for scalability. When we initially planned out our project, we did not know how many visualization operations we would need the rendering module to support. We did not know how much time it would take us to get as far as we did. Thus, the rendering module was designed to be easily extensible and flexible.

Before starting the rendering module, I looked at the existing software code to see which parts I could most gainfully accelerate in hardware. From my initial analysis, I came up with the clear_buffer module and the draw_fft_bin module described below.

The clear_buffer module provides the most significant gain in the hardware implementation over the software implementation. In software, this is essentially an I/O-bound operation with the processor spinning inside a loop to write a single value to every pixel location. In hardware, we can do much better because the OPB bus overhead is avoided completely and four consecutive writes can be performed in a burst.

The draw_fft_bin module is another improvement from the software implementation. Instead of having the processor write every pixel in a column for every FFT bin, the processor now only needs to use the FSL interface to write the opcode for draw_fft_bin, the height, the colour and the FFT bin number of the column it wishes to draw. This reduces the potential 480 writes per column down to 4 writes per column.

With the basic hardware acceleration in place, my efforts turned towards implementing more interesting-looking visualizations. The simplest improvement that I could think of was to add fading effects. The way that this effect works is as follows: when the frame buffers are flipped, instead of clearing the back buffer with a single colour, the back buffer is “cleared” by copying the current front buffer into it. The newly copied buffer is

faded by subtracting its colour values so that it will approach black, which has a colour value of 0.

4.2 Software

When I added new hardware features to the rendering module, I also had to update the software to take advantage of it. It was fairly straightforward to replace the software rendering code with the hardware rendering code as only a small fraction of the code was actually responsible for drawing the screen. As a test, I made verified that both the hardware and the software implementations produced the same output when the same song was played. My initial rendering software was fairly simple and was done more to show that the hardware worked rather than to maximize the final aesthetic qualities.

5. Problems encountered

By design, the rendering module was quite detached from the rest of the system. It receives commands from the FSL and it communicates with the outside world using the ZBT RAM controller. As such, I did not have to worry about all the external timing problems that plagued my team members.

The most troublesome problem I came across was from the implementation of fading effects. Initially, with fading the back buffer, the plan was to read directly from the front buffer and write the resulting faded value into the back buffer. I reasoned that the VGA ran at nearly half the speed of the memory interface and that with sufficient buffering, the VGA controller and the render module should both be able to access the same RAM module without being bandwidth limited.

Unfortunately, I was wrong. With my first implementation, the fading worked but was incredibly slow. It would appear that the arbitration overhead of the ZBT RAM module multiplexing between the VGA controller and the RAM module was immense. It was never designed for a high-contention environment.

After much thought, I decided that the best solution would be to instantiate another ZBT RAM controller to serve as a “mirror” of the front buffer. Being a separate RAM chip, there would be no issues with memory contention. Whenever a write was made to the back buffer, the render module would make the same write to the mirror buffer. Thus, when the back buffer is flipped to the front buffer, the mirror buffer would contain the same contents as the front buffer. When it came time fade the front buffer into the new back buffer, the render module could have exclusive access to the mirror buffer while the VGA controller has exclusive access to the actual front buffer.

Thankfully, this solution worked and we arrived at the fading implementation that is used in the final project.

6. Tools Used

In this project, I have mainly used the Xilinx Platform Studio tools and ModelSim. With XPS compilations taking upwards of 20 minutes to complete, ModelSim functional simulations must have saved me countless hours of debugging. With a good simulation, I was mostly confident that the design I program onto the FPGA would be working or at least very close to working.

The other tool that I found useful was the Xilinx Microprocessor Debugger (XMD). With the ability to read from the OPB bus using XMD, I found it useful to create a debugging OPB interface in the render module and map some signals to be output on the OPB bus. This is extremely helpful in debugging situations when you really have no idea what is wrong. I find this solution also a lot simpler, albeit more primitive, than using the Xilinx ChipScope tool which could be cumbersome when all you need to see are a few signals.

7. Community Contribution

I have not contributed to the bulletin board.

8. Feedback to Xilinx

I felt that the compilations took rather long to finish, even for the smallest changes. It would save a lot more time if there were some incremental compilation function where only changed portions of the design were recompiled.

9. Course Feedback

I enjoyed learning about the functionality of ModelSim. This project made me realize that it is a much more powerful simulation tool than the other simulation tools I have used in the past; namely, Quartus' simulation tool.

Finally, this design project made me realize that a lot of the time in hardware design was spent in debugging. Without a good simulation tool such as ModelSim, debugging would have taken longer than it did. I learned that if you simulate, the code may work; however, if you do not simulate it, the code probably will not work.