Team Haxorus

FPGA Music Visualizer

Gabriel Samaroo

Kyle Verma

Jonathan Johnson

Contents

Team Haxorus releases its code to anyone seeking to use it for educational purposes. Please include the following with any code used:

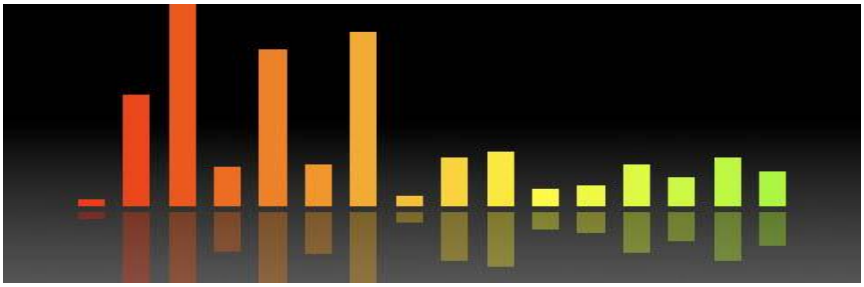
```
/*************************************************************************
* Authors:
*
*   Team Haxorus
*     - Gabriel Samaroo
*     - Kyle Verma
*     - Jonathan Johnson
*
* Notes:
*
*   Code developed Fall 2011 at Carnegie Mellon University
*   for use in 18-545 (Advanced Digital Design Projects).
*
* Contact Info:
*
*   Gabriel Samaroo  - samuhru8@gmail.com
*   Kyle Verma       - kyleverma@gmail.com
*   Jonathan Johnson - blackwolf189@gmail.com
*
*************************************************************************/
```

Chapter 1

Introduction

For our 15-545 (Advanced Digital Design Capstone) project, this group decided to implement a Music Visualizer on an FPGA board. Based on personal experience and from discussing with friends, we realized that the biggest problem with current music visualizers is the lack of user interaction. We addressed this issue with the XBOX Kinect.

The XBOX Kinect is an advanced motion tracking video camera that provides the depth data of the image it sees. Using this depth data, our project allows users to interact with a screen and develop a better proprioception. By playing with our device, users can improve their balance, posture, and flexibility.



Chapter 2

Hardware Design

In this chapter, we provide an overview of the design and implementation involved with the hardware of the system.

2.1 Overall Design

The board used is a Virtex-5 LX110T.

System Inputs:

- 3.5 mm Audio in – MP3 Player
- Kinect via Serial Port from Computer

System Outputs:

- Audio out through headphones and/or speakers
- DVI output from Chrontel 7301C chip

System Communication:

The audio input is sent to the AC'97 Analog-to-Digital Converter (ADC).

The output of the ADC is sent to:

- AC'97 Digital-to-Analog Converter (DAC) to provide audio out for the system

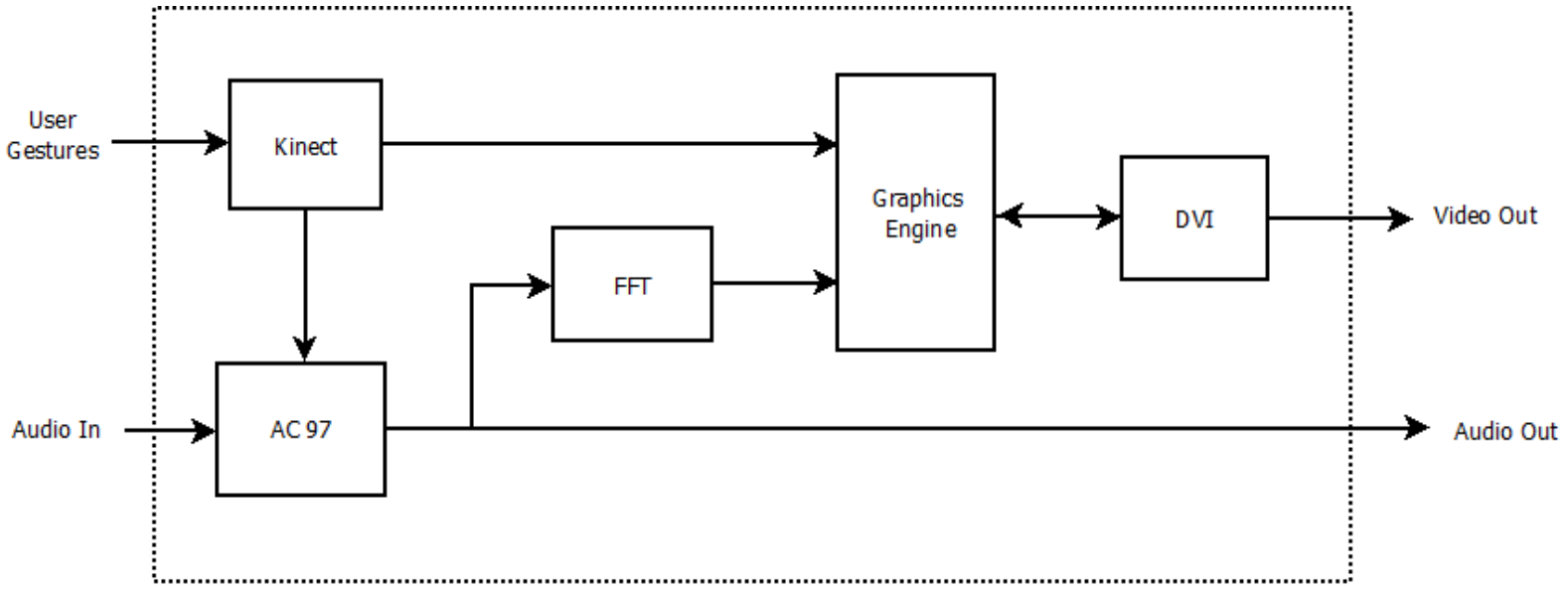
- Graphics Engine so that waves can be controlled by “loudness” of music (magnitude of audio out)

- Xilinx IP FFT – outputs real and imaginary components of audio music for different frequency levels. The magnitude of the real/imaginary components are estimated and sent to graphics engine along with the “frequency bin” index

The Kinect input goes through a series of gesture recognition modules to check for recognized gestures. When gesture is recognized, the graphics engine is signaled and carries out specific command. The Kinect also recognizes a volume up and volume down gesture and thus must signal AC 97 to alter volume levels.

The graphics engine gets an X and Y position from the DVI interface and sends back the pixel for that position. It also receives a switch_buf signal which allows the use of double buffering.

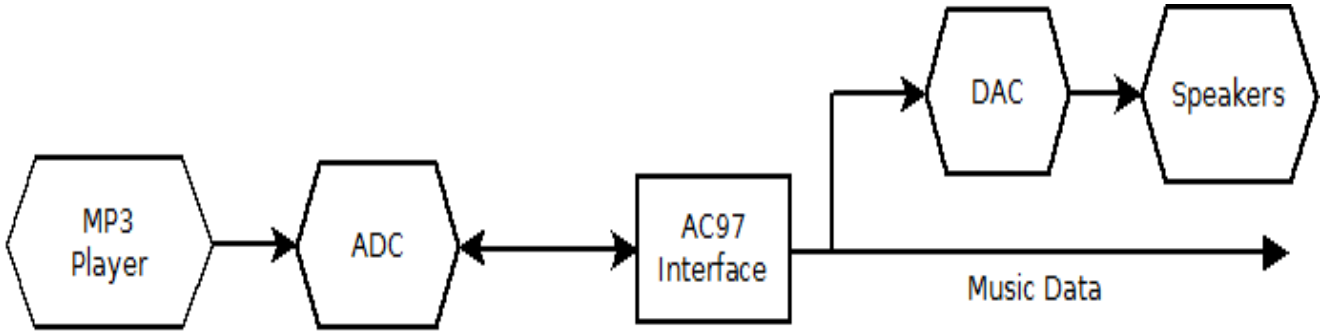
The DVI outputs data to the Chrontel 7301C DVI controller to display on an external monitor.



2.2 Audio

Audio is inputted to the board from an external device and fed into the AC'97 Analog-to-Digital Converter. The ADC mixes all the analog inputs to the board and outputs pulse-code modulation (pcm) left and right data. This data is sent across the AC Link from the AC' 97 Controller to the AC '97 Codec, which allows the audio to be outputted off the board.

Audio is sampled at the board bit-clock rate of 48 khz.



2.2.1 AC Link

The AC Link is the bridge between the AC' 97 Controller and the AC' 97 Codec. The communication consists of frames being sent across serially (msb first), with each frame containing 13 slots. The first slot is 16-bits and contains the TAG for the frame (marks which slots in the rest of the frame are valid). The remaining 12 slots are each 20 bits, making each frame 16 + 20*12 = 256 bits.

SYNC:

The AC Link generates a sync signal which tells Controller and Codec a new frame is beginning. The sync is held high for 16 clock cycles, during the transfer of Slot 0.

Slot 0 (TAG):

Tells Controller and Codec if frame is valid, and if so, which slots within the frame are valid.

Slots 1 and 2:

Slot 1 contains the stores the address of an AC 97 register. Depending on whether we want to read or write from these registers, the msb (bit 19) is set to 1 or 0. The least significant bits of this slot are filled with zeros.

Slot 2 contains either the data that we want to write or the data we requested to read during the previous frame. The data is 16 bits, so the least significant 4 bits are filled with zeros.

Slots 3 and 4:

These slots contain pcm left and right data from the ADC. This data is stored in registers for use in our system and sent across the link so the DAC can output audio off the board.

Slots 5 - 12:

These slots are currently being ignored, but can be used with some of the AC' 97's extra features, such as surround sound and microphone in.

2.3 Fast Fourier Transform

Xilinx's Fast Fourier Transform core was used to sync visualizations with the music. The pipelined architecture was chosen to allow for streaming input of audio data. The FFT output is unscaled, which requires more board resources, but allows for greater precision when doing calculations.

We decided to have a 12 bit index width, which creates 4096 "frequency bins". Because the AC 97 runs as a 48 khz clock rate, the range of frequencies in each bin is roughly:

$$\frac{48\ khz}{4096} \approx 11.7\ hz$$

The output of the FFT is two's complement, so we needed to take the absolute value of the real and imaginary output components. To calculate the magnitude from the real and imaginary components, we would normally take the square root of the sum of each component squared. Instead, we use the following magnitude estimate formula:

$$Magn_{est} = \frac{15}{16}\max(real, imaginary) + \frac{15}{32}\min(real, imaginary)$$

This formula accurately estimates the magnitude of the FFT output with just shift registers, comparators, and adders.

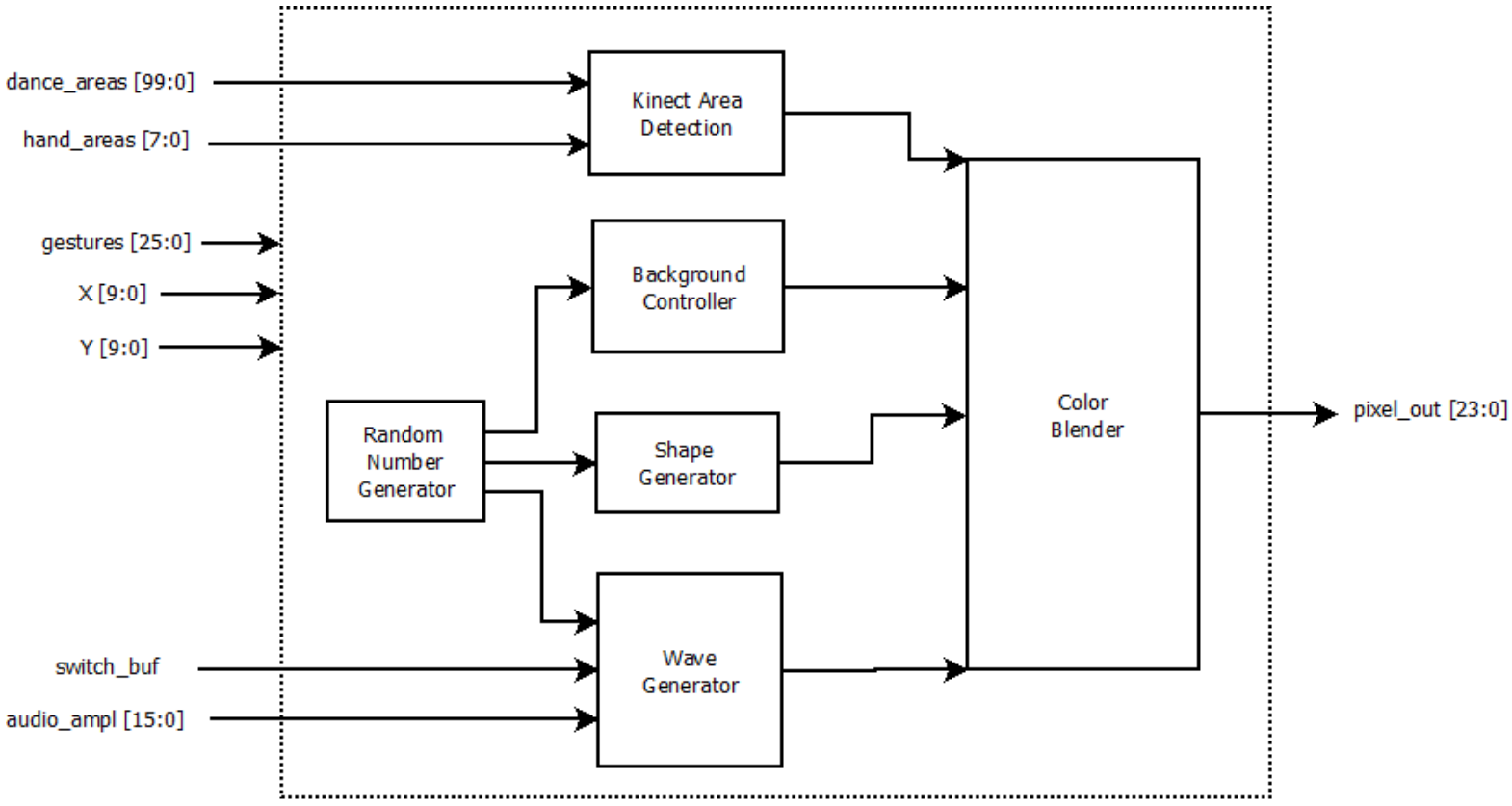
A done signal is sent out from the FFT every time the last index is reached, which signals the frequency waves to switch buffers (it uses double buffering).

2.4 Graphics Engine

2.4.1 Design

Components:

- ❖ Random Number Generator
- ❖ Background Controller
- ❖ Color Blender
- ❖ Shape Generator
- ❖ Wave Generator
- ❖ Frequency Wave Generator
- ❖ Kinect Area Detection



Random Number Generator:

Several 8-bit Linear Feedback Shift Registers (LFSR) are used to emulate a random number generator. There are a total of eight in the generator, each of which is seeded differently. The seed is determined by the length of time the user holds reset when initializing the device. Also, the registers are loaded with new data after a certain number of clock cycles. This generator is used to allow randomness in colors, amplitudes, speeds, locations, widths, and radiuses.

Background Controller:

Initializes background to random color and gradually changes red, blue, and green color components over time. The background color's strongest component (R, G, or B) is shaded horizontally and its second strongest component is shaded vertically.

Color Blender:

Takes in pixel color and valid signal for each the graphical component. Pixels are weighted differently, to differentiate between objects (ex: shapes are weighted stronger than background but less than waves). For each color component (R, G, B), all valid pixels for that component are averaged together with Xilinx IP Dividers. The output of this module is a blended pixel color for the current X/Y position.

Shape Generator:

Creates either circles or squares on screen. These components are random and unrelated to the music. Each shape has a random location and a horizontal and vertical direction of motion. If either the left or right wall is hit, the shape's horizontal direction is changed, and likewise with the top and bottom wall for vertical direction. To create circles, Xilinx IP Multipliers to calculate a circle's algebraic formula: $X^2 + Y^2 \leq r^2$

Kinect Area Detection:

When areas of screen detected as being active, these modules alter the color of that area so that user knows what they are doing.

Wave Generator:

Each wave has a random speed and color. The amplitude of each wave is controlled by an average of a certain range of the FFT output. The width of each wave is controlled by the loudness of the music (amplitude of the audio straight out of the ADC). Registers are used to keep track of a wave's phase (up motion, down motion, constant motion).

Because there is only one Y location for every X location in a wave (passes Vertical Line Test), waves coordinates are stored in a block ram. Each block ram has a data width of 10-bits and an address of 10-bits. The X coordinate gives is used to address the ram and the Y coordinate is stored as data.

Motion is created by shifting all the entries in a block ram over by one and writing to the newly freed address. To do this, the *double buffering* technique is used, in which one buffer is read from while the other is written to. As soon as the buffer being written to is complete, the two buffers switch so that the newly written buffer will be displayed and the old one will be written over.

Frequency Wave Generator:

These waves display the frequency output from different ranges of the audio data. The double buffering technique is used for these waves as well to prevent flickering of visualizations.

Frequency Ranges Used		
Bass	**Mid-Range**	**High-Range**
60 hz – 250 hz	250 hz – 2,000 hz	2,000 hz – 6,000 hz

2.4.2 Communication

The Graphics Engine takes in a 10-bit X and Y coordinate and a switch_buf signal from the DVI interface. The X and Y coordinates allows the graphics to calculate an output pixel from the background color and the shapes/waves valid at that location. The DVI interface has a 50 mhz pixel clock and the Graphics Engine runs twice as fast at the 100 mhz system clock. This allows the graphics engine to spend one cycle reading from memory and a second cycle to compute a blended output pixel.

The Kinect sends gestures to the Graphics Engine which can alter visualizations. The following changes can occur:

- Randomize background color
- Invert background color
- Check Background color
- Randomize wave 1, 2 or 3
- Change type of wave 1, 2, or 3
- Turn shapes on/off
- Change shape type
- Randomize shapes

2.5 DVI

The DVI is initialized with the I2C serial protocol. The protocol communicates with the Chrontel 7301C chip on the board and sets registers according to our design specifications.

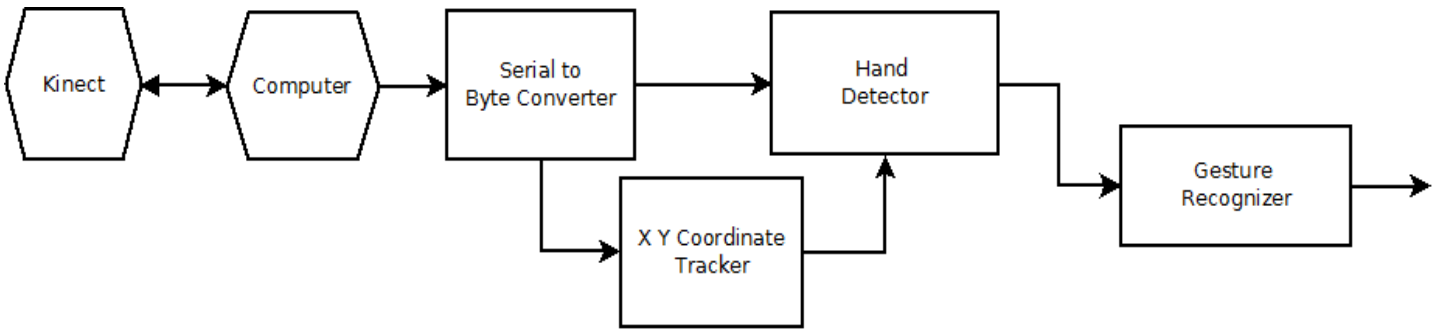
A sync generator sends the Horizontal and Vertical sync signal to the external monitor. In addition, the current X and Y position are kept track of and outputted to the graphics engine.

Since our resolution is 640 by 480, a switch_buf signal is sent every 307,200 pixels.

2.6 Kinect

2.6.1 Description

The Kinect is actually a USB hub that contains a motor, an accelerometer, an RGB camera, a depth camera and a microphone. For this project we only paid attention to the depth camera. However to get the Kinect up and running you have to activate the motor. Once the motor is activated you can interact with the Kinect as you would any USB hub. When a read request is sent to the depth camera it sends back a frame of depth pixels. One frame is 640 x 480 pixels. Each pixel is a number that ranges from 2047 to 0 with smaller numbers representing objects that are closer to the Kinect.



2.6.2 USB

The Virtex 5 board contains the cypress CY7C67300 USB host controller. However the documentation for this controller is extremely terse and does not describe well how to use the available pins on the board to actually interact with any USB device. Additionally, some of the documentation seems to indicate that it doesn't work with USB hubs and only support USB 1.1 speeds. For these three reasons we decided fairly early on that USB was not the way to go if we wanted to finish the Kinect part of the project by the end of the semester.

2.6.3 Serial

We decided that by using a serial cable, we could use a computer to allow the Kinect to interact with the board much faster than trying to get the USB driver working. The serial interface on the board is simply a single pin that follows the normal serial protocol:

- the line is held high until one byte is sent across
- the line drops to zero to signal the start of a byte
- the byte is sent across the pin (least significant bit first)

The ease of this protocol allowed us to quickly get the Kinect sending depth data to the board, which allowed us to focus on interpreting the depth data and getting gesture recognition working in a much smaller time frame.

Despite its ease, there are two major limitations when using the serial port with regard to the Kinect:

1. The maximum speed we can operate our serial connection at is 500,000 bits per second.

   ➔ In an attempt to solve this problem we only send one quarter of the depth pixels in any given frame, however even with this fix it took about one second to send an entire frame of depth data across the serial connection which in turn means that it takes about a second for a gesture to be recognized.

2. The serial connection sends one byte at a time where each pixel of depth data is 11 bits.

   ➔ To deal with this problem, we only send the 8 most significant bits of every pixel. The problem with this solution is that we do not get as accurate of information as we would have liked.

2.6.4 Python

We used a python script to interface with the Kinect and the serial connection to the board. To do this, we needed the following python libraries: pySerial, freenect, python-numpy, pyton-dev and python. Because of the way the depth camera works, it is impossible for the Kinect to recognize anything closer than a certain poin, therefore the number zero is not a valid or possible depth pixel. Taking this into account the script would grab a frame of depth data, divide the entire frame by 8 and send every fourth pixel across the serial connection. Then, the script would send a zero byte across the line to signal the end of the frame, and start the process over again.

2.6.5 Design

Gesture Recognition (1 month):

To do gesture recognition several modules needed to be created. It is easiest to understand if each module is described first.

Serial Byte Reader (2 weeks):

The serial byte reader is fairly self explanatory; it monitors the serial input line. When this line goes low, a byte is coming across, so the byte reader fills eight registers with the byte that has come across the line. Byte valid is then held high for one clock cycle of the system clock. The serial line sends data far slower than the system clock so the byte reader is set up to read the data at the far slower rate of the serial input.

Pixel Position Monitor (3 days):

This module monitors bytes as they come across the serial line. It keeps track of an x and y position so other modules can easily know the x and y position of the byte that has just come across.

G Monitor (5 days):

This module monitors pixels as they are read and sent to the board. If the pixel is within the x and y ranges supplied to this module, and lower than a threshold value controlled by switches on the board, it will increase pixel_count. Once pixel_count reaches a preset threshold (parameter), the module holds asserts that area as active until the same area of pixels comes across the line again.

Final Gesture Recognition (6 days):

G Monitor is set to monitor eight equally sized portions of the screen. A gesture is simply checking to see if two of the G Monitor modules is holding gesture out when byte zero comes across the line.

2.6.6 Gestures

		Shape	
Randomize	**Change type**	**Power on/off**	

Randomize Wave			
Wave 1	**Wave 2**	**Wave 3**	

			Wave Type
Wave 1	**Wave 2**	**Wave 3**	

	Background		
Randomize	**Invert**	**Checker**	

Volume Up			**Volume Up**
Volume Down			**Volume Down**

Chapter 3

Miscellaneous Notes

3.1 Tools

- Xilinx ISE →Used for synthesis, translating, and programming the board
- ChipScope →Used for Verifying and Debugging
- VCS → Used for simulation

3.2 AC Link Bug – Line In Register

One of our longest delays was due to setting an incorrect value in an AC 97 register. When trying to get the FFT to work, we assumed the 16 bits of left and right data coming out from the AC 97 was automatically valid because the speakers were outputting the correct audio data. This caused us to spend a lot of time messing with the FFT because the output always seemed random and unrelated to the music. It wasn't until a while after when we noticed the audio data was always a small number and not quite right.

We learned that we needed to set the line in register to mute in order for the audio data to come out of the ADC correctly. We're still not sure why this needs to be done, and the datasheet doesn't mention this, but it fixed our problem.

Chapter 4

Overview

4.1 What Went Wrong

4.1.1 SystemACE Interface

Using the SystemACE CompactFlash Solution datasheet, an interface was written that reads from the CompactFlash device. This was eventually dropped before it was completed because we had been spending a lot of time on it and realized it wouldn't actually be useful for our project.

4.1.2 Memory Interface Generator

Roughly 1.5 weeks were spent trying to get the MIG to work. Eventually, we realized that the MIG was not optimal for our design and switched to using block rams. The fact that the MIG requires burst reads/writes made it sub-optimal because we wanted to alternate between reads/writes for our graphics engine. In addition, getting even the example design to run was very difficult due to problems with the constraint file.

Two tips we can give to anyone using the MIG that help are:

- There are tutorials online that take you through the steps of using the MIG, but make sure you select the right chip part when running CoreGen – the part can be seen on a label under the board
- Manually enter the pin constraints in CoreGen – trying to create your own ucf file is extremely difficult

4.1.3 Temporal Pattern Recognition

Perhaps the biggest failure of Team Haxorus was the inability to implement a temporal pattern recognition algorithm as we had planned to do when beginning our project. This circuit was a simplistic one, consisting of multiple comparator circuits, and was left to be completed only if time permitted.

Currently, our synthesizable code includes the circuitry necessary to compare the output from both the AC '97 analog to digital converter, and up to three filters connected to the adc (high pass, band pass, low pass). We were unable to incorporate this into our final demonstration solely due to lack of time implementing machine learning functions to match the patterns we wanted to discover.

In the future, the output of these comparators will be passed into a machine learning program to discover patterns that represent musical features. These features include crescendo, decrescendo, beat drop, and beat build up. By visually inspecting the WAV file of music, a person can easily discern each of these characteristics. Since WAV files are a temporal representation of the music, it is only a matter of time and RAM to enable this feature within our project.

4.2 What We Could Improve

- Connect XBOX Kinect to board via USB. This would get rid of the computer and allow our system to run entirely on the board. It would also allow huge improvements in gesture recognition.

- Provide better scaling / modifying of FFT output to make visualizations sync better with music.

- Add microphone to system. This would be very easy being that mic_in is just controlled by registers in the AC' 97.

- Add in more pattern recognition. The pattern recognition code was left out of our design, so it would be nice to get it working. This would allow us to detect more musical characteristics (ex: crescendos beat drops, beat patterns, etc.).

Chapter 5

Sentiments

Kyle Verma

For this project my major duties were getting the AC ‘97 working properly and getting the Kinect working. The entire group worked at different points on the AC ‘97 which turned out to take up much more time than anyone had predicted, however since our project is a music visualizer it was vital everyone knows how the AC ‘97 functions. When it came time to split into our separate parts, I was forced to put my part on hold and make sure the AC ‘97 worked properly because video was a more important component of the design than the Kinect interface. After the setbacks due to the AC ‘97 and the incredible difficulty in figuring out how the Cypress USB host controller works I made the decision to use a serial cable in conjunction with a computer running a python script to get the Kinect communicating with the board. Once information was actually sent across the serial line it still took quite some time to figure out the maximum baud rate that our system could work at and the correct slowdown required to grab the information sent over the serial cable. This was all completed shortly after Thanksgiving break which was fortuitous because we realized there was a bug in the way our AC ‘97 interface was designed so me and Gabe were able to spend the last few weeks fixing that bug and making the video output look as good as possible.

In this course I thoroughly enjoyed the freedom granted to students to pick a project and run with it. However FPGA programming presents serious problems with getting anything working, even something as simple as communication using a serial cable took several weeks and the data sheets were really clear about how the serial communication works. Figuring out how to get the more complicated things on the board such as AC ‘97 is a daunting task, and figuring out something that isn’t well document such as the Cypress USB Host or ACE controller isn’t something that is particularly plausible in a one semester capstone class.

Overall I feel that we picked a particularly good project because we were able to achieve all of our deliver-ables in almost exactly the time frame of the class which felt good. I do regret the group problems that we encountered because we could have had a more polished product at the end if we weren’t essentially a two man team but pattern recognition was the most removable part of the project so it was lucky that that was the teammate that flaked out on us. I feel that this class was a great learning experience on how to work in a group, how to deal with working on a large group project and how to work with an FPGA. I only have two complaints about the class: The lab computers and environment do not work as easily as they should and there should be better tutorials on how to get the students started working in the environment provided for this class.

Jonathan Johnson

Team Haxorus viewed this class with very different eyes than most other groups saw it as. We decided early on that this was a class to demonstrate our capabilities with an FPGA, and hence we never chose our design as a 'project', but instead as several interfaces we would each develop and link together. This proved to be a great decision, as I required little knowledge of what my partners were working on, and only needed to know the interfaces they expected from me.

Through out this project, the most trying and difficult exercise was to maintain faith in our idea. Other groups had the ability to measure their progress, as most attempted to build a system that was already in existence. When you have an existing product to measure your progress against, it is relatively easy to determine how much work is left for you. However, Team Haxorus only had an idea to go on, and nothing to measure our progress against. Having faith in my teammates turned out to be quite trying at times and yet our product turned out better than I could have ever conceived.

It is important to note that each team member has a part of our project that they personally own; a design that no one else was allowed to make recommendations on. It is important that each individual in a group feel their importance and their ability to influence the group, even when they are outvoted during times of dispute. The emotional attachment to your work is crucial; it builds pride, a sense of accomplishment when things begin to work, and motivates an individual to stay in lab long past team meeting hours. It was mentioned by another group that, after over hearing our argument, they believed we would soon be enemies after this class ended. Nothing could be further from the truth. The fact that each of us argues so passionately makes it clear to all others that we care about the work we have done, and will nearly bite our own teammates head off to defend our work. Passion drives ambition in my opinion.

My last thought falls on how simplistic our design was. Once again, my team merely linked together an audio, video input and video output interface. It was up to each individual to make their system work, and criticism only came after we linked our code together and did not get the results expected. Although it was a risky decision, I am glad we never commanded anything of each other, other than our interface ports. This allowed great flexibility in our work, and the ability to cut out parts of our project as deadlines approached.

I greatly enjoyed this class, and I see a great future for our work.

Team Haxorus: If you are hacking, you are not us.

- Jonathan Johnson

Gabriel Samaroo

Let me start off by saying this was by far the most I've ever worked in any single Carnegie Mellon class, and as result, I learned a great deal about team dynamics, using an FPGA, debugging, the importance of developing and keeping up with a schedule, and just myself in general. I started the semester having several courses worth of experience with Verilog, but knowing very little about FPGA's. In addition, I didn't really know anyone in the class, so my team assignment was technically random. My teammate Jonathan came up with the idea of a Music Visualizer and it sounded cool, so I decided to go along for the ride.

Throughout this class, I wasted large chunks of time due to inexperience. My first three weeks were spent working on reading from the CompactFlash. While this isn't very difficult, learning how to use/program and not really understanding how to use chipscope caused me to spend a lot of time not really accomplishing anything. Later, I spent roughly 1.5 weeks working with the MIG before I realized it wasn't the most efficient memory to use for our design. Finally, roughly 2 weeks were spent working with the FFT before realizing the issue was with the audio data coming out of the AC 97. This was almost a black box for me because Kyle and Jonathan had written it and music was being outputted correctly, so I just assumed it worked. This time wasted helped to teach me some very important lessons. First off, I now understand how to use an FPGA and what resources to use depending on what I'm trying to do. More importantly, however, I learned to never assume someone else's code is 100% correct and to make sure to fully understand anything I include my project.

I became quite good at using chipscope and I **highly** recommend anyone taking this class to learn to use it early on because it can become your greatest tool. You should generate an ILA and ICON with a large data bus and just set the data bits you're not using to 0's. Once you have chipscope running, you can do things like combine bits to view as a bus and change the format to things such as hex, binary, unsigned decimal, signed decimal, etc.

I believe everyone in our three person team contributed differently to the project. One of my partners was great at getting us organized early on and made sure we clearly outlined what we were doing so that we could track our progression. At the same time, I think this partner did *very* little in respect to the actual project itself and hurt the group more than he helped. This partner constantly missed lab and the biggest problem was that when he did show up, he was not ready to work. We constantly argued throughout the semester and his response to any dispute was that he had more experience, therefore anything he says trumps whatever we say. This experience has taught me to recognize someone that I feel is not a good teammate early on so that I can do something about it as opposed to ignore it and do all the work myself.

My contribution of actual code was to write the entirety of the Graphics Engine, the FFT, and rewrite the AC Link from scratch. While I feel that I wrote a large majority of the code, I think my second partner helped me with everything I did. We worked together to debug and fix the AC Link in a short amount of time. Also, a large portion of the graphics engine is to use trial and error (speeds of waves, color shading, controls, etc.) and this partner gave me great ideas on what to implement and feedback on what he liked/disliked.

Overall, this course was a great experience and I hope our project and experiences can help other people in some way.