EE109 Final Project Report

Travis Geis & Axel Sly
6 Jun 2015

Introduction

Our EE109 final project is a real-time audio visualizer implemented using a combination of software and hardware. In this document, we discuss our project goals, the implementation details, challenges we encountered, and the results of our project.

Goals

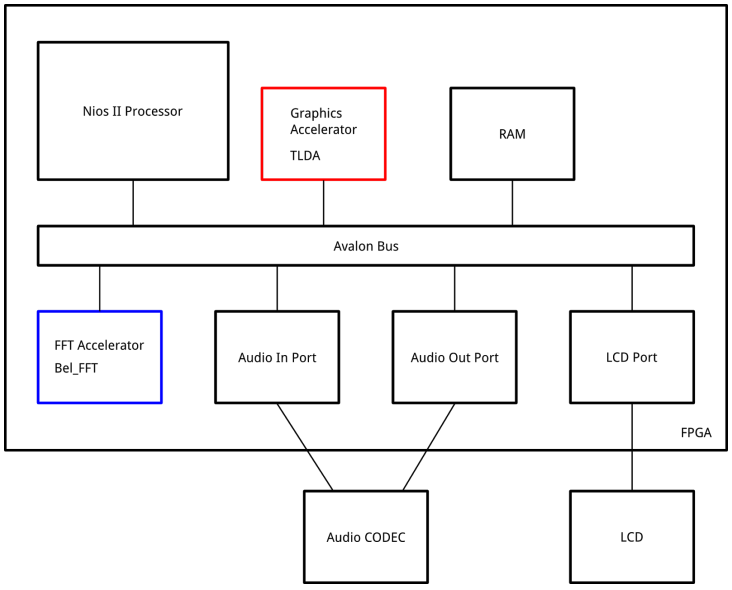
The overarching project objective was to create a graphical representation of audio that reacts to changes in the audio in real time. This goal leads to three sub-goals:

1. to capture audio,
2. to analyze the spectral information in the captured audio, and
3. to draw a corresponding animation.

These three goals determined our system architecture, as we discuss below.

System Overview

The audio visualizer system is a Qsys project based on the ee109-media-computer from Lab 2. The relevant hardware is shown in the following diagram:



The system includes a Nios II processor, RAM, and the Altera Audio and LCD Port peripherals connected to the Avalon bus. In addition, the system contains a fast fourier

transform (FFT) module and a custom graphics accelerator, the thick-line-drawing algorithm or "TLDA" peripheral. These additional components also connect to the Avalon bus as memory-mapped peripherals. Below we detail each significant component of the system.

FFT Accelerator

Our project uses a hardware FFT module to accelerate the computation of the FFT. It computes the FFT of a full array of data, then draws a visualization using the result. Therefore, the system needs an FFT implementation that operates on a block of data rather than a stream. Initially, we considered many different ways to take the FFT, including:

1. A software function,
2. a custom hardware module,
3. the Altera FFT Megacore,
4. two different cores from opencores.org,
5. and the BelFFT project.

We wanted to compute the FFT more quickly than possible in software, so we decided to look for hardware solutions early in planning the project. We researched how to create a custom FFT module in hardware, and decided that making our own FFT system would be too difficult in the time we had for the project. The Altera FFT Megacore documentation was very complicated, and we were not convinced that we would be able to use the Megacore even if we had the license to do so.

We looked on opencores.org for simpler FFT modules, and we found two candidates: one from Unicore Systems and one from Gisselquist Technologies. These modules were configured to process data streams, and did not include Avalon interfaces. To test the modules, we wrapped them in custom finite state machines that managed streaming the data to the modules from an array, and streaming the results back to memory, so the cores could process arrays of data rather than streams. Unfortunately, while we verified the functionality of our FSMs in testbenches, neither of the downloaded FFT modules gave correct results when run on known datasets. We had to reject both FFTs after spending a significant amount of time setting them up for testing.

Luckily, after extensive further searching, we found the BelFFT project (http://sourceforge.net/projects/belfft/), which provides a Java-based verilog generator for creating hardware FFT modules. BelFFT can produce FFT cores with 16- or 32-bit input values, over an arbitrary number of points. We chose to generate a 128-point FFT with 32-bit inputs. These inputs are two's-complement signed integers, which conveniently match the output format of the audio module. BelFFT generates an Avalon interface and testbench for the module, so we were able to simulate the

complete FFT module before adding it to the Qsys project. The module worked correctly once integrated into the Qsys project.

TLDA: Graphics Accelerator

Our project also uses a hardware graphics accelerator, which implements a “Thick Line Drawing Algorithm” (TLDA). The TLDA draws straight lines of arbitrary thickness. At the top level, the TLDA circuit is a finite state machine, similar to the finite state machine in the LDA_circuit of Lab 2. This state machine follows the line designated by the two points provided by the user. However, instead of drawing a pixel at each point along the line, the TLDA calculates the endpoints of a “perpendicular” line starting at the current point and whose length is the user-provided thickness, and uses the LDA_circuit to draw the perpendicular line. Due to the complexity of computing a truly perpendicular line using integer or fixed-point arithmetic, these lines are not actually perpendicular to the main line; instead, they are aligned to the axis that is most nearly perpendicular to the main line. Thus for a nearly-vertical line, the cross-hatching lines are aligned with the x-axis, and for a nearly-horizontal line, they follow the y-axis. This approximation of perpendicular makes the TLDA simpler.

We enabled our TLDA peripheral to take a user-specified address and thickness. The user specified address is what the TLDA circuit uses as the base address to draw the pixels in memory. Allowing for different base addresses enables us to indicate the location of the current backbuffer to the TLDA, so our graphics can be double-buffered. The thickness parameter specifies the thickness of the line that the TLDA draws.

Audio System

Central to our project is the ability to take in audio samples and mirror them to the output, so that audio can pass through to speakers. We began by writing a simple audio driver for our platform. The driver uses the Altera Audio module’s registers to configure the audio system, and binds the audio module to a user-specified callback function. When the audio input FIFO is 75% full, the callback runs, and the user can copy the audio data into memory.

We configured the hardware in Qsys such that the line-in port leads to the input of the ADC on the audio codec, and the output of the codec’s DAC leads to the headphone port. The audio codec samples at 48 kHz, so the maximum resolvable frequency (the Nyquist limit) is 24 kHz, which is adequate for capturing all audible sound. Although 48 kHz is a relatively high sampling rate for audio, it is far slower the our system’s 50 MHz processor clock, so we concluded that managing the audio data completely in software would not be a challenge on our platform. There is adequate time to copy audio from the input FIFOs to RAM and then back to the output FIFOs without missing the audio timing window.

In order to take an accurate FFT on a buffer of audio, all samples in the buffer must be contiguous in time. In other words, the buffer must be filled with consecutive samples, then remain unmodified while the FFT module performs its calculations. If the buffer contains more than one fragment of the audio signal, then it will likely contain sharp edges that create high-frequency noise on the output of the FFT.

To provide the FFT module with a suitable buffer of audio, our system uses three buffers for audio: the left and right live audio buffers, and the FFT sample buffer. The FFT function runs synchronously in the main loop of the software, and at the top of the loop, we set a flag indicating to the audio interrupt service routine that the FFT is ready for new samples. On the next audio interrupt, the service routine copies the samples into the live audio buffers and into the FFT sample buffer, then signals to the main loop that samples are ready. Finally, the main loop proceeds to the FFT function and then to drawing the results.

Challenges

We encountered many challenges in implementing our project, not least of which was finding a suitable FFT module. Once we had found a working FFT module, we then needed to integrate it into the Qsys project, and we initially had trouble doing so. The Qsys output directory did not match the Quartus input directory, and it took us several hours to discover why our project was not building with the correct files.

We also had several bugs in the TLDA module. For example, if the input line had zero length, meaning its start and end points were identical, the TLDA would still draw a single row of the line. We fixed this bug by adding a simple check to the start state of the TLDA. In addition, we forgot to assign the X and Y values of the TLDA in one of its state machine states. The result was correct in the testbench, but in the compiled hardware system, the lines would generally have every other row missing. We had to read the source code very carefully several times before we discovered the cause of this bug, but it was trivial to fix once we found it. Finally, the TLDA still erroneously draws a single pixel at the origin of the screen for every line it draws. We were unable to determine the cause of this bug.

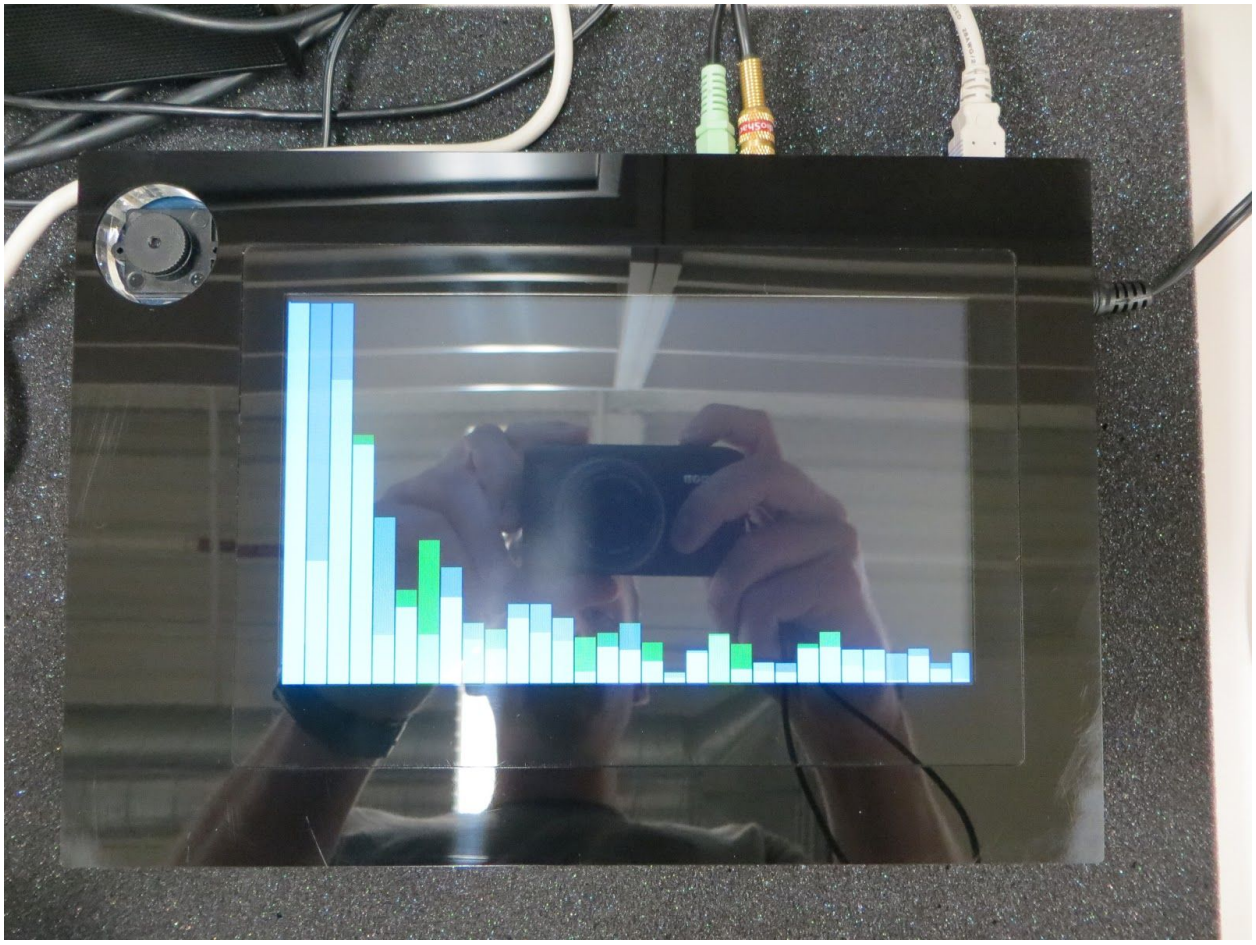
Currently, we are not scaling the results of the FFT before displaying them, so the lower-frequency components dominate. Ideally, we would apply a logarithmic scale before drawing, so each frequency component has equal weight. However, we we not satisfied with the speed of computing a logarithm in software on our system, so we opted to leave the values unscaled and have a more responsive graphic.

Because we used the Lab 3 starter files are the basis for our project, our system contains many superfluous components. However, we were unable to remove these extra components without breaking the functionality of the starter files, so we opted to include them.

Results

Our music visualizer works correctly, and shows a responsive graphical representation of the spectrum of incoming audio. The system has a working audio driver, with interrupt functionality. It performs the FFT and the majority of the drawing operations in hardware.

Below is a photo of the output of the audio visualizer:



Appendix A: Using the Project

To use the audio visualizer:

1. Load the project final.ncf in the Altera monitor program and download the system to the DE2-115 development board. Though the monitor program claims we are using trial versions of Altera Megafunctions, these are remnants of the Lab 2 starter files, and are not necessary for the functionality of our system. You may click “”disconnect” on the time-limitation window.
2. Compile and load the software, then run it.
3. Connect an audio patch cable from an audio source to the blue line-in port on the board.
4. Connect the green headphone jack on the board to speakers or headphones.

The sound should be audible on the output device and the audio visualizer should react to the sounds. If the bars seem “stuck” at high values even when no sound plays, pause and then run the system in the Monitor program. This symptom seems to be related to the debugging interface.

Appendix B: File Hierarchy

The project files are divided at the top level into hardware and software.

- In the **hardware** folder are the top-level module (ee109-meia-computer.v) and Quartus project file (ee109_media_computer.qpf), along with the synthesized system (ee109_media_computer_time_limited.sof).
    - In the **hardware/tlda** folder are the files for the TLDA, including its testbench files.
    - In the **hardware/belfft** folder are the FFT core files generated by BelFFT.
- In the **software** folder are the Monitor program’s **final.ncf** file and the **src** subfolder.
    - Inside **software/src**, main.c contains the system’s main software, with globals defined in system_globals and various ISRs in their respective files.
        - In **software/src/belfft** are the software drivers from the BelFFT project.
        - In **software/src/kiss_fft** are software FFT routines that we used for testing.
        - In **software/src/tlda** is the simple driver for the TLDA module.
        - In **software/src/ee109-lib** is our custom audio driver.

Works Cited

EE109 Lab 3 Starter Files.

Schofield, Tim and Adrian Wong. “Real Time Spectrogram project.” Cornell University School of Electrical and Computer Engineering, 2007. Accessed 8 Jun 2015 <https://courses.cit.cornell.edu/ece576/FinalProjects/f2007/tjs49aw259/>.

Storm, Frank. “BelFFT: FFT co-processor in Verilog based on the KISS FFT.” 14 Dec 2013. Accessed 8 Jun 2015 <http://sourceforge.net/projects/belfft/>.