VOCODER PROJECT

ECE532 Digital Hardware

March 28, 2005

Brian Chan 990835389
Simon Deleon 990956408

Overview

Introduction

Vocoders are used to compress audio information in multimedia and communication systems. Specifically, vocoders are typically used to encode voice information by modeling speech so that the salient features are captured in as few bits as possible. They are also commonly used in musical applications to achieve synthetic, robot-like sounds from acoustic sources.

Channel vocoders are a subset of vocoders that achieve compression by breaking down the signal content into frequency components. Analog vocoders use sub-band filtering (the process of applying a bank of band-pass filters to the audio signal) to achieve this frequency decomposition. Because voiced sounds can be approximated by sinusoids, a periodic pulse generator recreates voiced sounds. Since unvoiced sounds are noise-like, a pseudo-noise generator is applied, and all values are scaled by the energy estimates given by the band-pass filter set. A channel vocoder can achieve an intelligible but synthetic voice using 2,400 bps.

The real-time channel vocoder we developed works by analyzing the frequencies in the modulator (voice) signal, splitting them into bands with a fast fourier transform (FFT), finding the magnitude of each band, and then amplifying the corresponding bands of a carrier signal by that magnitude. Typically the carrier will be a frequency-rich signal such as noise.

System

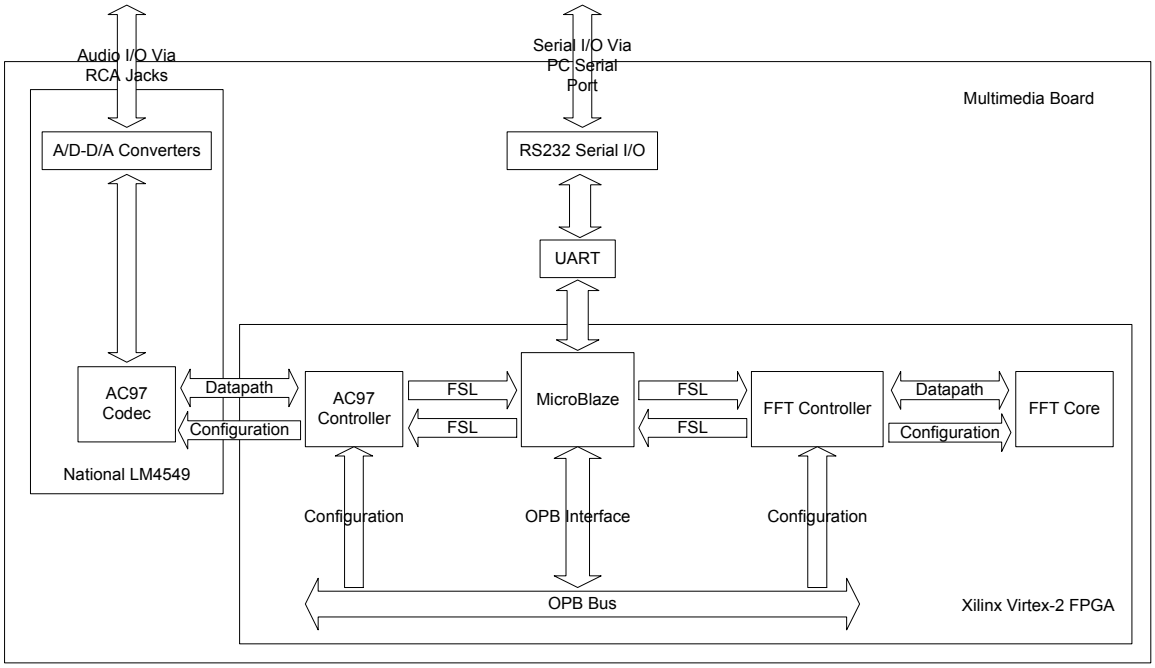

Figure 1- System Block Diagram

In order to create our vocoder, we needed to use the AC97 Core (from the AC97 Project, ECE532 2004) to record and output audio, FFT Core (from CoreGen, XFFT_v3.1) to transform audio data into the frequency domain, UART to communicate text to the monitor, and FSL and OPB instances to communicate and control the cores from the Microblaze.

We developed the FFT controller to send/receive data from the FFT core via FSL, and to latch control and configuration signals from the OPB Bus. Since the FFT Core works in burst mode, the input FSL to the FFT Controller must be filled with the same number of samples as the FFT point-size. For example, when performing a 512-point FFT, the user application must transfer 512 audio samples onto the FFT's FSL input. The FFT Core will begin calculation and automatically write the results back to the FSL bus once the appropriate command is written onto the OPB Bus at the address of the FFT Controller. This command specifies the scaling schedule, point size, and direction (forward/inverse) of the FFT operation.

The user application ('system.c') achieves vocoding by multiplying the coefficients of the left and right audio channels. Typically, the user will input voice on one channel and white noise on the other. Please see outcomes and future work sections for suggestions on improving the application software.

Outcome

The streaming FFT and IFFT operations work flawlessly on streaming audio at the standard CD quality sampling rate of 44.1 kHz. Recovered audio from the transform operation sounds identical to the source material when input levels are adjusted appropriately (to avoid clipping from loud sounds). The vocoder operation is over-zealous and attempts to modify the entire carrier spectrum by the spectrum of the modulator. For example, the vocoder operation will multiply all 512 coefficients from a 512-point FFT, which is unnecessary to achieve the intended effect. Typical vocoders used in musical applications use only 8/16/32 channels (to achieve data compression). As a result, our vocoder operation results in a voice that sounds like a robot talking under water. It's a very interesting effect but it is unnecessarily computer intensive (the coefficient multiplications are performed entirely in the Microblaze) and is not the intended effect.

A very simple fix to achieve 32-band vocoding with the available software is to simply drop all but 32 channels in the transform of the modulator. So, for example, using a 128 point FFT we would preserve 32 points (in the frequency range of voice) and discard the other 96 coefficients.

Future Work

Now that the FFT works perfectly, many other frequency domain functions can be performed in addition to vocoding, such as equalizers, frequency shifting/pitch alteration

etc. Extension to 2D FFT for video applications will require a reduction in the depth parameters of the FSL if a second FFT is instantiated, since our design already uses up almost all of the BRAM on the FPGA. Since the audio sounds better when using a larger FFT (a 1024pt FFT sounds better than using a 64pt FFT), we would recommend reducing the size of the FSL to the AC97 Core. Multiple reads from the AC97 could be used to fill up one FSL to an FFT (since the FSL must be at least as large as the number of data points used by the FFT).

Block Description

AC97 Codec

We used the core and drivers from the AC97 Project (2004 ECE532 project) to record and output audio. We changed the *init_sound* function in the *AC97.c* driver to use the RCA audio jacks in order to receive two independent channels of input (see the LM4249A datasheet for the required register values). The AC97 core that we used had an FSL interface for data transfer from the Microblaze and an OPB interface for control. We noticed that when we used the FSL to get large amounts of data from the AC97, the FSL depth had to be set to twice the actual sample size to get good sounding audio (so if we were trying to get 512 samples at once, the FSL depth had to be 1024). We aren't sure why this occurs, but it's important to note (this also puts constraints on the size of the FSL depth, since we were running short on BRAM when using large FSL depths).

FFT Controller

We used COREGEN to instantiate the XFFT_v3.1, 2048pt Radix-4 burst mode FFT core within our FFT controller. This core had a bug for CoreGen v.6.3i. This was fixed by patching with the appropriate service pack. In order to integrate the core we had to create VHDL wrappers for a controller that communicated with FSL and OPB. The OPB had control signals to start the FFT operation, as well as specify scaling, direction (FFT or inverse FFT), and size (of the N-pt FFT). The data to and from the FFT core was transferred through two FSL paths. The VHDL controller managed the connections between the various components and achieved proper timing through the use of two state machines. See Figure 2 for an illustration of these connections.

The OPB state machine and interface handler are responsible for initiating an FFT calculation and configuring the FFT core beforehand. FFT configuration values for length, scaling schedule, and direction (forward/inverse) are latched from the OPB data bus when a chip select signal is raised.

The FSL state machine will begin emptying the contents from the FSL input bus when the OPB state machine has completed configuration of the FFT core. Results of the calculation are immediately written back to the FSL output bus and made available to the Microblaze processor.

We created a test-bench for the FFT controller, with a model of the FSL and OPB behaviour. This test-bench was used to exhaustively test the FFT before we tested in hardware. We should have implemented the FSL and OPB models using the actual cores

instead of modeling the behaviour ourselves, since we ran into a number of problems due to incorrect assumptions about the FSL/OPB behaviour. We fixed all the problems with the test-bench and it now accurately models the FFT Controller and communication with the FSL and OPB.

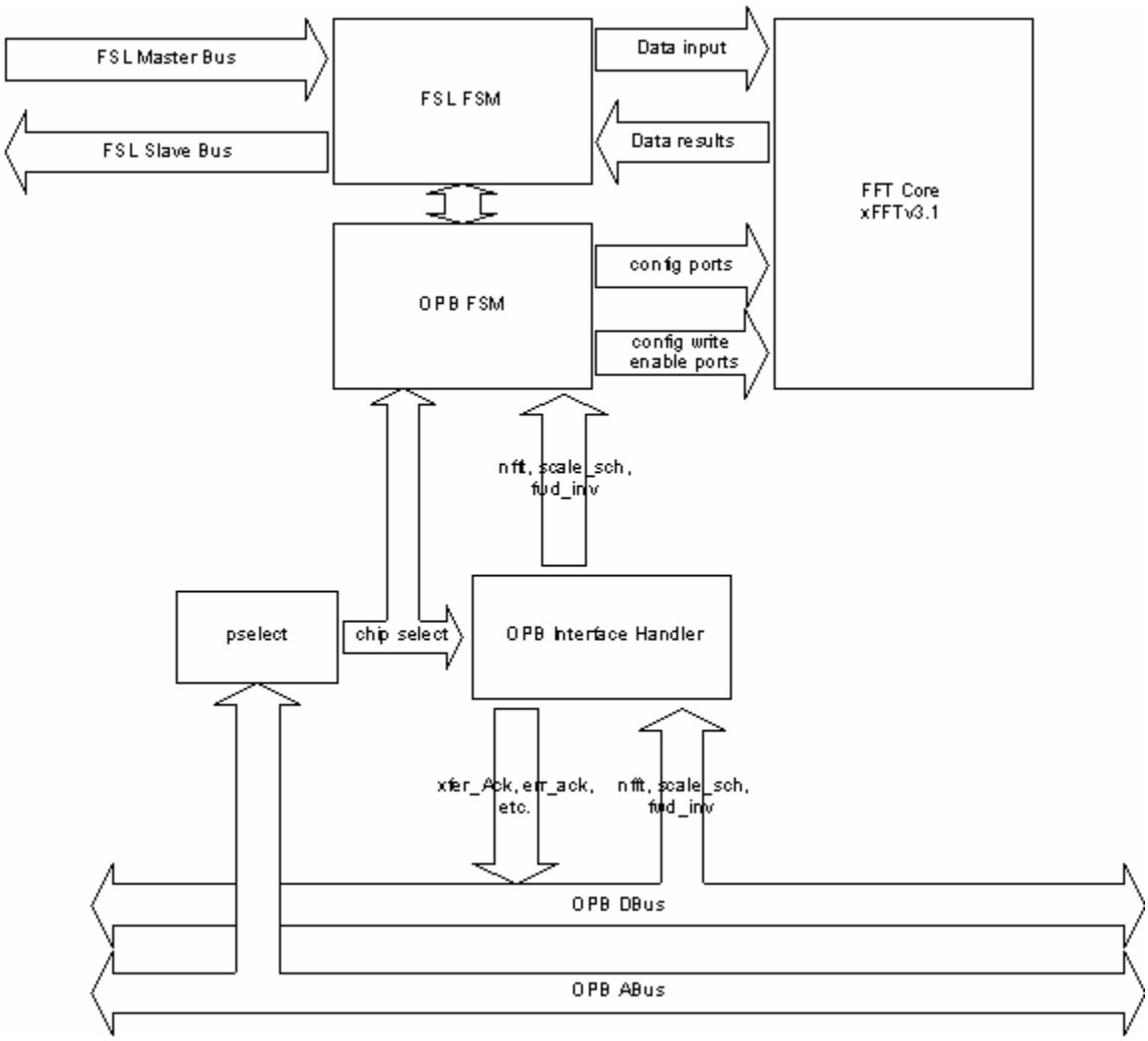

Figure 2 - FFT Controller Functional Block Diagram

FSL

One set of FSLs was used to communicate with the AC97 core and another set was used for communicating data to the FFT. The FFT Controller that we wrote and the AC97 controller from the AC97 Core managed data transfer onto and off the FSL. We created a Modelsim test-bench and test-vectors to verify the operation of the FSL and determine the correct timing to use in communicating with the FFT controller that we built.

OPB

The OPB was used to send control signals to and from the AC97 core and FFT core.

UART
The UART was used for debugging purposes, and can also be used to display text from software.

Design Tree

1) hw - contains rtl, testbench, and Modelsim scripts. Also contains a pcore directory that can be dropped into other XPS projects.

2) sw - contains system.c source file for implementing vocoding using fft core.

3) ise - contains 'hw' contents in ise project 'fft_controller', ready to run presynthesis simulation by typing 'presynth.do' within project directory from Modelsim prompt. Another project entitled 'fsl' was the ise project we used to verify the operation of the fsl links used in the vocoder.

4) xps - contains 'sw' contents and 'pcore' contents in xps project format, ready to run in fft/ifft audio mode with vocoding off.

5) doc - contains final report