FPGA Implementation of Radix-2 FFT Algorithm for an Equalizer with Custom Shield

Ram Tripathi

Deen Dayal Upadhyaya College (University of Delhi)

Email: ram.22hel2231@ddu.du.ac.in

Abstract

This paper serves to implement the Cooley-Tukey Radix-2 Algorithm targeted for Spartan 7 family Arty s7-25 FPGA having Arduino/Microblaze pin layout. In the realm of Digital Signal Processing (DSP), the Fast Fourier Transform (FFT) serves as an efficient and basic block. The software version though easy to implement is constrained by the sequential nature of execution owing to the limitations of processor instruction throughput. In contrast, the hardware FFT implementation (using digital logic and Field programmable gate arrays) with its parallel processing capabilities helps to achieve exponential improvements in throughput capabilities over software FFTs executed in DSP microprocessors.

This article aims to demonstrate a practical implementation of the highly efficient Cooley-Tukey algorithm of DSP with Xilinx Spartan 7 family's Arty S7-25 board as the target for designing a custom shield for the Arduino/Microblaze form factor pin layout of the board and thus serving as a robust educational and real-life application setup to showcase the Hardware Acceleration Capabilities of the Field Programmable Gate Arrays.

2

Description

1 Overall Architecture

The macroscopic architecture of the system is described in Figure 2.1. The core of the system consists of the FFT, gain curve applier, IFFT, and the gain curve block ram. To enter a gain curve, the user can select one of the available input methods, two of which are unimplemented at this time. Two smaller RAMs are used to interface between the main system and the display subsystem, which operates on a 65MHz clock instead of the system's standard 27MHz clock.

2 Audio Subsystem

Figure 2.2 describes the main part of this system, the audio manipulator that applies a gain curve to the audio stream.

2.1 AC'97 Layer

An AC'97 codec chip handles the multiplexing and conversion of audio data into a digital format. A provided module handles the serial communication with this chip and a wrapper layer simplifies the signal control. [2]

Essentially, only three signals are used between this layer and the rest of the system. The **ready** signal is asserted high for a single clock cycle when the AC'97 chip can send and receive data. This occurs at the sampling frequency, 48KHz. At this point, the signals **from_ac97_data** and **to_ac97_data** can be used. The former is a signed 20-bit output containing an audio sample and the latter is an input for a signed 20-bit value.

The AC'97 wrapper layer is essentially the same as the one used in the 6.111 spectrograph class example, [3] but with a few modifications. When Switch 2 on the Labkit is on, the module will forward all **from_ac_data** directly to **to_ac97_data** without going through the audio modification subsystem; otherwise, **to_ac97_data** is the output of the equalizer system.

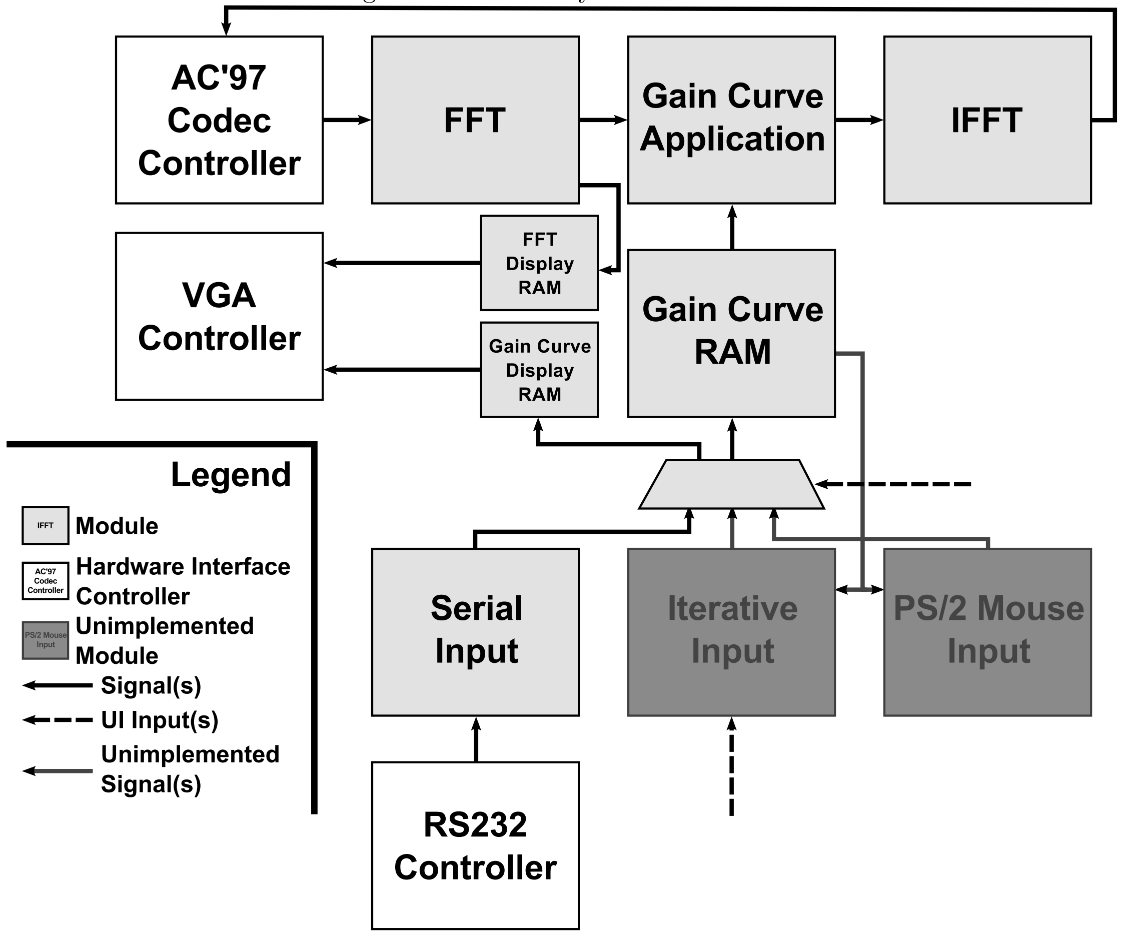
Secondly, when Switch 3 is on, **from_ac_data** is the output of a 750Hz sine wave that was generated and stored in memory. When it's off, Switch 6 controls whether to get input from the microphone (when the switch is off) or from the line-in RCA plugs.

2.2 FFT

The Fast Fourier Transform algorithm efficiently transforms a time-domain signal into the frequency domain. That is for an N-point FFT, it yields X_k from the input samples X_n in the following manner:

$$X_k[k] = \sum_{n=0}^{N-1} X_n \exp\left\{\frac{-2\pi j}{N} n \cdot k\right\} \tag{2.1}$$

Figure 2.1: Overall system architecture



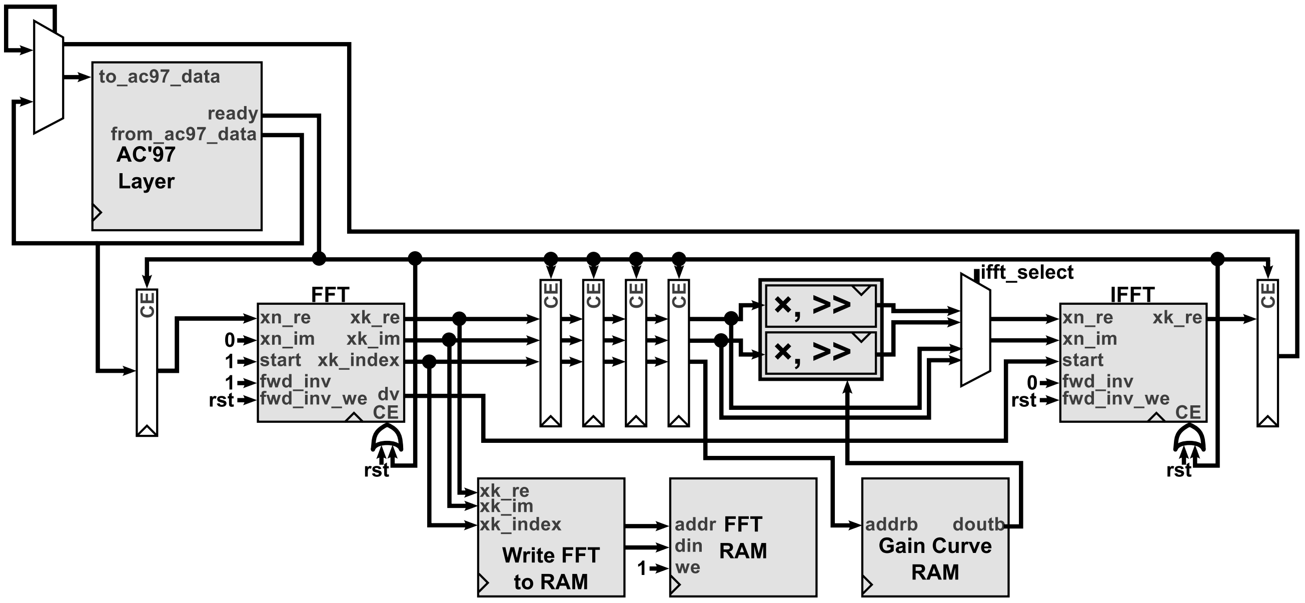
An N-point FFT generates N frequency "bins," values that hold some range of frequencies based on the sampling frequency. In this system, a 1024-point FFT was used and the incoming audio signal was 48KHz, so each bin held $\frac{48000}{1024} \approx 47$ Hz. In general, the lower the bin's frequency coverage, the better; therefore, it is advantageous to choose an FFT with as many points as possible.

The FFT module for this lab was generated with the Xilinx CoreGen wizard. The following options were chosen in the wizard:

- 1024 points
- pipelined/streaming mode
- 12-bit input
- 12-bit phase factor
- unscaled operation
- use clock enable (CE) pin
- use natural ordering

The data entering the FFT was a buffered **from_ac_data**. The pipelining of this signal, which was only allowed to change when **ready** was high, preventing garbage data from entering the FFT.

Figure 2.2: Audio subsystem



The **ready** signal was also extremely important when using the FFT. If the FFT processed data faster than the AC'97's clock, it would generate results too fast from the same set of input samples, and the effect would be as though the audio waveform had slowed down (yielding an audio waveform analogous to the result of applying pressure to a spinning vinyl on a turntable). Therefore, the clock enable pin was only high on **ready**.

The FFT also has a few other control signals. **start** is always enabled, **fwd_inv** is always high to indicate that the module is to perform a forward transform. This data is latched, so **fwd_inv_we** is high when the system performs its power-on reset and off otherwise.

The real and imaginary components of X_k are forwarded to the gain curve application module as well as a simple module that writes them to RAM for live display on the screen. This latter module uses the value from **xk_index** to determine the address in the FFT RAM. The **dv** (data valid) signal is necessary for correctly timing later modules.

2.3 Gain Curve Application

The outputs of the FFT (the real and imaginary components of X_k) is fed into a series of registers to delay the signals. The motivation for this delay will be discussed in 2.4.

After the real and imaginary components have been delayed, the resulting signals enter the gain curve application phase.

The index of the FFT output (which has also been equivalently delayed) is fed into the gain curve RAM and the real and imaginary components of the FFT are multiplied by the output of that RAM. This is a clocked operation, but it's clocked to the 27MHz system clock, so from the perspective of the far slower, 48KHz clock, the operation is instantaneous. A more accurate implementation would pipeline the result of the multiplication by a 48KHz clock, but the output would be indistinguishable from the user's perspective.

After the multiplication, the components are divided by 256, the maximum value of the gain curve (they are actually arithmetically shifted by 8, an equivalent operation). As a result, all frequency values are effectively multiplied by a gain less than one since the maximum gain is 255. This prevents "clipping," an effect where the output speaker voltage is pushed to its max and the speaker is overdriven.

The output of this system is sent to a multiplexer, where the user can use Switch 7 to select whether to use the gain-corrected output of the FFT (off) or the pure output of the FFT (on) as the input into the inverse transform for the final processing phase.

2.4 IFFT

The inverse Fourier transform, as its name suggests, changes a frequency-domain signal into the time domain. Given a frequency domain signal of X_k, we an obtain the time domain signal X_n as follows:

$$X_n[n] = \frac{1}{N} \sum_{k=0}^{N-1} X_k \exp\left\{\frac{2\pi j}{N} k \cdot n\right\} \tag{2.2}$$

It is important to note that the forward transform described in 2.2 and the inverse transform described here are nearly identical. The FFT algorithm can be easily modified into an IFFT by taking the complex conjugate of the inputs and applying a $\frac{1}{N}$ scaling factor. The Xilinx FFT module supports an inverse transform mode when **fwd_inv** is set to 0. It does, however, require more work to get the transform to operate correctly.

The same core generated in 2.2 was used to compute the IFFT. To simplify the system a little, consider the design case where a user wishes to redirect the output of an FFT into an IFFT. In an ideal scenario, the signal going into the FFT and the signal coming out of the IFFT will be exactly the same. Of course, because sampling is discrete, small errors will arise; this noise can be reduced by increasing the size of the FFT.

Because the FFT and IFFT are in unscaled mode, the input of the IFFT must be pre-multiplied by $\frac{1}{N}$, as the Xilinx core does not do this. This can be accomplished simply by performing a right arithmetic shift by $\lg(N)$.

Secondly, the IFFT shouldn't start until the FFT is producing valid output data. There are several output signals from the FFT that indicate the validity of the data, namely **dv**, **done**, and **edone**. The first of these is most effective because it goes high as soon as the FFT starts producing data and remains high thereafter. Therefore, we can feed **dv** of the FFT into **start** of the IFFT. This guarantees that the IFFT will always be synchronized to the start of the FFT.

The output of the FFT must be in natural ordering. Due to the way that the FFT algorithm functions, the frequency coefficients do not normally appear in order; a RAM and re-ordering algorithm must be instantiated in order to correct this problem.

Lastly, the inverse FFT needs three cycles after the start signal before it can accept data. Therefore, the output of the FFT needs to be appropriately delayed using the 48KHz **ready** signal as its clock. This is accomplished by using registers with a clock enable line set to **ready**.

The real output of the IFFT is fed into a register, also clocked to **ready**, which is then sent to **to_ac97_data**.

3 Display Subsystem

The display subsystem consists of a VGA controller and two modules that specify the value of a pixel given its location and data from RAM. All of these modules operate on a 65 MHz clock, created with the FPGA's DCM.

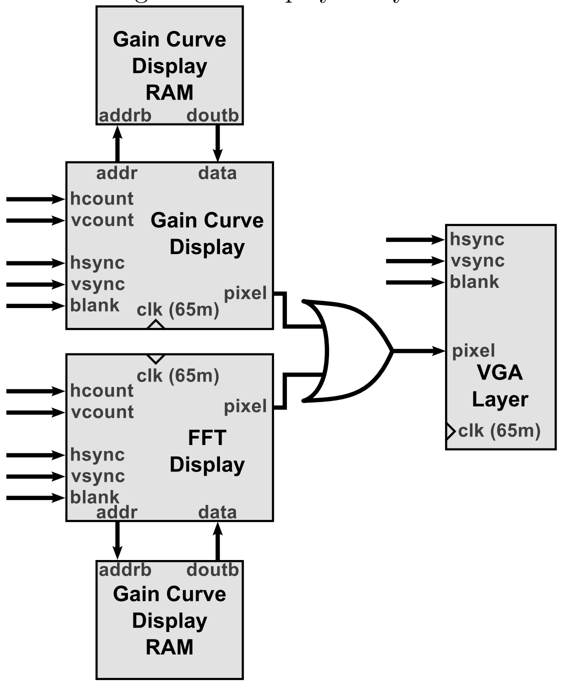
The VGA controller generates timing signals: the horizontal synchronizer **hsync**, vertical synchronizer **vsync**, and blanking signal **blank**. It also outputs the current position of the VGA scanner through **hcount**, a 11-bit value describing the current horizontal position, and **vcount**, a 10-bit value for the vertical position.

The screen resolution is 1024×768 and the screen is split into three sections to show the current gain curve, the absolute value of the FFT's real component, and the absolute value of the FFT's imaginary component.

3.1 Gain Curve Display

(see Appendix 6)

Figure 2.3: Display subsystem



The gain curve display module reads data from the gain curve display RAM by assigning the RAM address to the current horizontal value. It then paints a pixel a positive color if the current vertical value is between the bottom of the graph and the value of the gain curve at that address.

3.2 FFT Display

(see Appendix 5)

The FFT display shows the absolute values of the real and imaginary components of the FFT in the same manner as the gain curve display. To compute the absolute value of a component, the signal is tested to see if the most significant bit is 1. If it is, according to two's complement arithmetic, the signal's value is flipped and 1 is added to it.

4 Input Subsystem

This device provides a framework for developing many methods for the user to input a new gain curve. Currently, only the serial input method is functional, but the framework for an iterative input algorithm was built and a mouse-based input system was also planned.

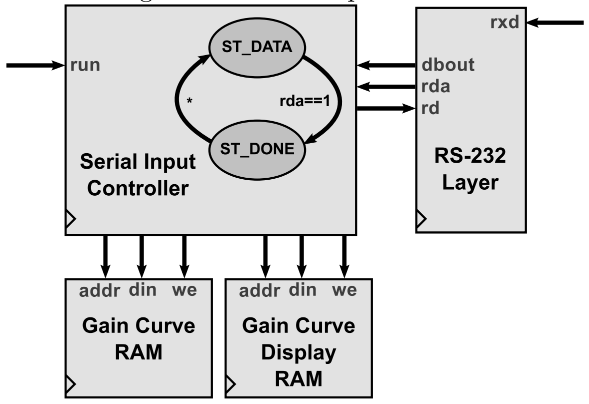
4.1 Serial Input

(see Appendix 7)

A user can connect a computer to the RS-232 port via a null-modem crossover cable and transfer a gain curve. A switch is toggled to start the process. Next, the user transmits a gain curve at 9600bps, with one stop bit and odd parity, and one byte per FFT bin. Therefore, the first byte that the user transfers will contain the gain for the first bin, the second will be for the second bin, etc.

Figure 2.4 describes the serial controller. The main module resets its internal state machine to its initial state (*ST_DATA*) and sets the RAM address counters to 0 when **run** is first asserted. The state machine interacts with the RS-232 protocol layer, grabbing a byte of data whenever one is available. When it receives a byte, the controller stores it to the gain curve RAM as well as the gain curve display RAM. If the number of FFT points, N, is less than or equal to the display width, 1024, the controller increments the addresses of both the gain curve RAM and the display RAM. If N is greater than the display size, the controller will only update the address of the display RAM every $\frac{N}{1024}$ bytes. For example, for an 8192-point FFT, the display RAM's address will only be updated once every 8 bytes received. This is accomplished by keeping an **index** counter and only taking the top 10 for the display RAM address signal.

Figure 2.4: Serial input method



The RS-232 protocol layer was designed by Digilent Corporation for their Nexys2 Spartan-3E evaluation board [4] (see Appendix 12). This component receives data from the RX serial port and generates a byte by oversampling the data and looking for transitions between bits as well as checking the odd parity bit. It places this byte in a buffer.

A slight modification to the module was necessary to get the correct baud rate. The divisor is determined by dividing the clock in the following manner:

$$\text{divisor} = \frac{clockfreq}{\text{baud} \cdot 16 \cdot 2} \tag{2.3}$$

The Nexys2 board has a clock speed of 50MHz and the Labkit has a clock speed of 27MHz, so the divisor was changed to 89 to account for the change in clock frequency. Higher baud rates can be used for faster transfer speeds.

The interface between the gain curve input and this RS-232 layer simply reads data from a buffer and then asserts a signal indicating that it read that data, yielding a two-state state machine as shown in Figure 2.4 and described in Table 2.1. The **rd** signal is asserted when the serial control reads the byte from the RS-232 layer.

Table 2.1: Outputs of serial input state machine

State	**State Name**	**rd** (Completed Data Read)	**Other Actions**
0	ST_DATA	0	Save **dbout** to the gain curve and display RAMs
1	ST_DONE	1	Increment **index**

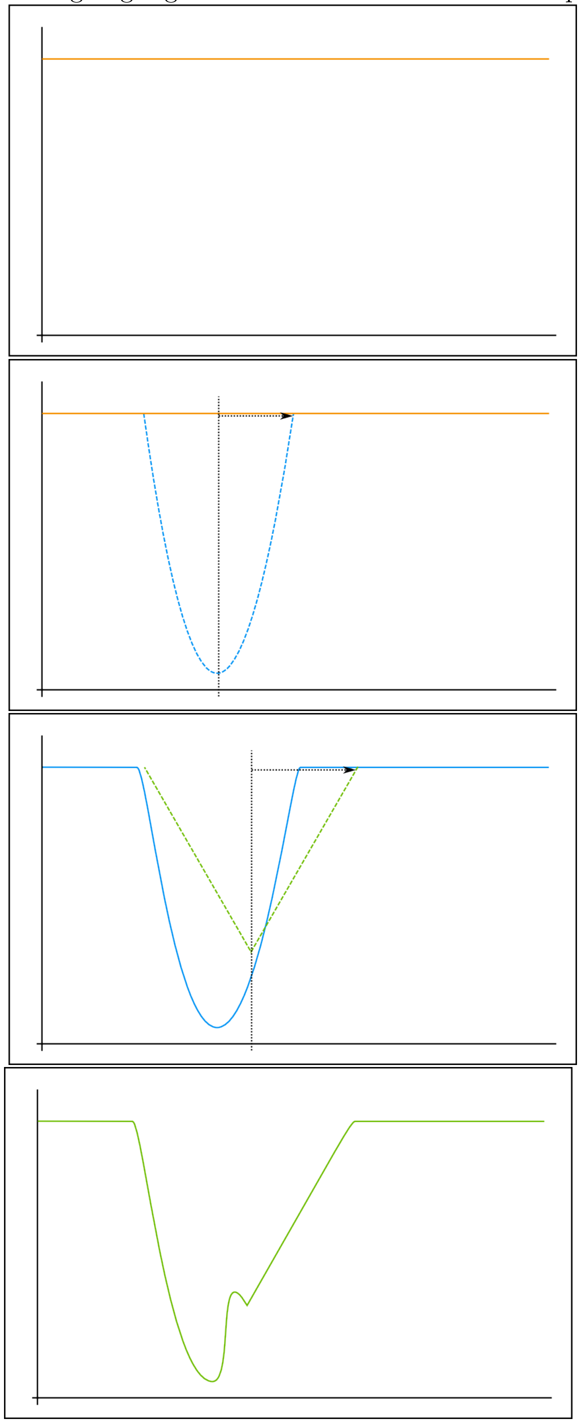
4.2 Incremental Input

(see Appendix 8)

The incremental input system provides an interface for the user to design gain curve directly with the FPGA using buttons on the Labkit or perhaps a PS/2 keyboard. The concept behind making the input "incremental" comes from the parametric equalizer's overall design property as being an audio filter. If a user finds that a sample of audio has, for example, noise at some specific frequencies, he can easily remove them by implementing “notch” filters that remove them.

The principle behind the incremental input system is that the user will make a filter of some shape, height, location, and bandwidth and place it on the gain curve. Then, the user will make another one and the two filters will be composed together smoothly, as shown in Figure 2.5.

Figure 2.5: Designing a gain curve with the incremental input method



Unfortunately, while some code was produced to start the design of this system (see Appendices 8, 9, and 10), it wasn't completed due to time constraints.

4.3 Mouse Input

A third possible input method would be to draw a gain curve with a mouse, just as one would draw with a piece of drawing software. This facilitates easy manipulation of the audio waveform and can be utilized by users who wish to experiment with audio effects.

5 Testing and Debugging

While the concept of the device itself was fairly simple, the implementation of the FFT and IFFT turned out to be somewhat complex than originally anticipated due to their low popularity and therefore relatively little documentation. The FFT module was instantiated in the same way that the class example's FFT was instantiated. [5], but for the IFFT module, it was necessary to introduce delays into the input signal and synchronize the start time with the output of the FFT. The Xilinx datasheet [6] did not document the core particularly well and only parenthetically mentioned the need for scaling and delay of the input signal into the IFFT.

To verify the operability of the FFT-gain-IFFT section, a 750Hz sine wave stored in memory was fed into the FFT. The gain curve was bypassed so that the IFFT would receive the scaled data of the FFT. At first, the FFT wasn't created with natural ordering enabled, so the output of the IFFT sounded nothing like the input. The effective result was that the output frequency bins did not match up with their expected input frequency bins of the IFFT.

The **done** signal of the FFT is asserted once every N computations, so this signal was used as a trigger on a logic analyzer. Also displayed on the analyzer were the outputs of the FFT, the magnitude of IFFT, and the FFT index. The expected result of feeding a 750Hz sine wave into the module was a single bin in the first half of the FFT with a large value (possibly with some surrounding harmonic bins containing very small values due to fact that only an infinitely large transform would have just a single bin with a nonzero value).

After ensuring that the FFT was correctly performing its computation and that the IFFT was being fed a scaled input, the magnitude of the IFFT output yielded very strange results. Essentially, the tone generated was a composition of two sine waves of equal magnitude and 90-degree phase shift. This problem was due to the fact that the IFFT input signal was not delayed correctly. To fix the problem, a variable delay system was introduced to determine exactly how far to delay the IFFT input. Once this delay was correctly applied, the sound output matched the sound input.

When switching from the sine tone to the output signals from AC'97, a full-spectrum noise (a "popping" or "clicking" sound) was introduced. Due to time constraints, the problem wasn't resolved, but it was narrowed down to having occurred somewhere between the AC'97 module and the FFT. It is likely that a timing issue causes the FFT input to temporarily become invalid, introducing noise into the calculation. The noise itself is not very noticeable on the magnitude output of the IFFT, but it can definitely be heard by the human ear.