FPGA Implementation of the Fast Fourier Transform

Garrett Massman and Cory Walker

December 2, 2014

Introduction

In this project we implemented a fast Fourier transform algorithm on a Xilinx FPGA using VHDL. The primary motivation of our project was to digitally analyze musical signals, but the use cases for an FFT chip far surpass that specific use case.

As a very general overview, our completed device functions by sampling an analog signal for a set amount of time and storing the data into an input buffer in BRAM. Next, the signal is is processed using an Fourier Transform Controller and stored in an output buffer. Finally, a microcontroller can then read the output buffer over a standard SPI protocol. From there the processed frequency domain data can be sent to a computer for further processing or any other device.

Motivation

The FFT is a very common operation used in many digital signal processing tasks. Digital signal processors are usually very highly tuned to perform a series of dedicated tasks quickly, and have very little flexability in the way of general purpose usage. This is both a blessing and a curse, as they rarely need to be reprogrammed, but are also limited to the range of tasks they can accomplish. High end DSPs utilize parallelism to complete mathematical calculations quickly. Because of their dedicated nature, DSPs also tend to be rather costly, upwards of $32 for a SHARC model chip from Analog Electronics.

FPGAs are primarily a large collection of configurable logic blocks that can be wired together to run in parallel, so why not try and use one in place of an expensive DSP? By taking the heavy computations off of a typical CPU and putting the burden onto an FPGA, the processor could be freed up to execute any number of instructions until the heavy computations are complete. This hybrid model has actually become quite popular in the past several years, and several products such as some of Intel's Xeon chips have FPGAs integrated onboard. This can offset the costs for companies who formerly may have been inclined to build expensive ASICs and instead shift their focus onto a new platform. A parallel CPU/FPGA integrated system is

beyond the scope of this project, but instead a simple model using a Digilent Basys board and an Arduino are built to illustrate the principles.

Theory

To understand the discrete Fourier transform, one must first analyze its analogous continuous time form. This is written most simply as

$$X(j\omega) = \int_{-\infty}^{\infty} x(t)e^{-j\omega t}\,dt$$

which defines the transform of a continuous time signal $f(t)$. Without getting into too many details, it defines the signal as a combination of sinusoids, making it a very useful tool for real world applications.

The discrete Fourier transform (DFT) is simply a reduction of the continuous Fourier transform into a discrete sample space. In other words, if we let x_n represent a sampled version of the continuous time function $x(t)$ with a total of N samples, we can replace the integral with a summation over the series, as shown below.

$$X_k = \sum_{n=0}^{N-1} x_n e^{\frac{-j2\pi kn}{N}}$$

As with the continuous time case, this series gives us a glimpse into the component elements of our original signal. However, it is rather costly to compute, requiring a time complexity of $O(N^2)$. For each value X_k, a series of values from $n = 0$ to $N - 1$ must be generated and summed, using up valuable hardware resources. Thus, calculating the DFT in this way is very inefficient, so we instead turn to the Fast Fourier transform.

In order to perform this operation more quickly, we utilized the Cooley-Tukey FFT algorithm implemented through the Xilinx CORE Generator FFT module. One requirement is that our input is strictly a power of 2. That is because the algorithm works by recursively finding the FFT of smaller and smaller sample sizes of x_n, which arises from the fact that the transform itself is periodic. The transform function can be broken into the sum of its even

and odd components,

$$X_k = \sum_{m=0}^{N/2-1} x_{2m} e^{-\frac{2\pi i}{N}(2m)k} + \sum_{m=0}^{N/2-1} x_{2m+1} e^{-\frac{2\pi i}{N}(2m+1)k}$$

This equation can further be simplified by making the substitutions

$$E_k = \sum_{m=0}^{N/2-1} x_{2m} e^{-\frac{2\pi i}{N}(2m)k} \qquad O_k = \sum_{m=0}^{N/2-1} x_{2m+1} e^{-\frac{2\pi i}{N/2}mk}$$

giving us

$$X_k = E_k + e^{-\frac{2\pi i}{N}k} O_k$$

The expression $e^{-\frac{2\pi i}{N}k}$ is commonly called the *twiddle factor*. Furthermore, because of the periodicity of the transform, we can calculate respective even and odd components simultaneously

$$E_k = E_{k+\frac{N}{2}} \qquad O_k = O_{k+\frac{N}{2}}$$

System Overview

A critical task in building our FFT device was converting an arbitrary waveform into the SPI signal input that the FPGA expected. This requires the use of a fast analog to digital converter. The Xilinx Spartan-3E FPGA does not come with an onboard ADC. Because of this, a large percentage of the work was performed by electronics external to the FPGA:

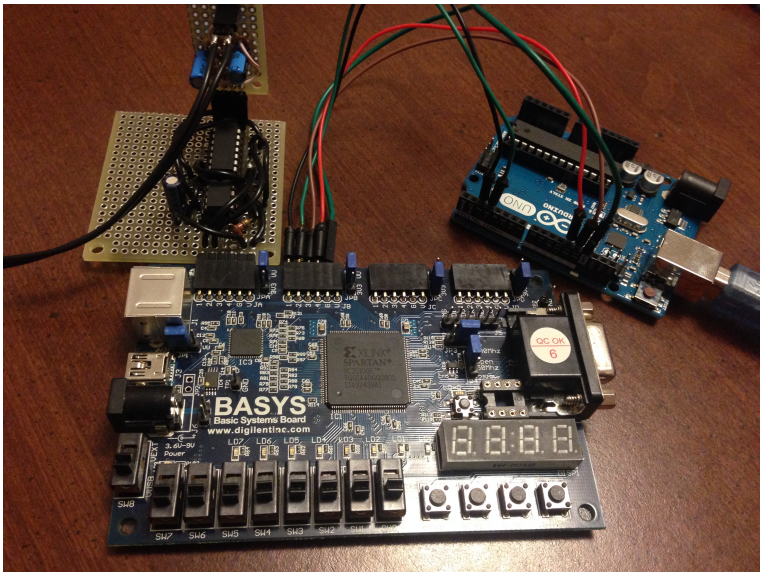

Figure 1: Full system including ADC PMOD and audio signal module.

These external components condition the signal and then send the digital signal into the FPGA at the right voltage level. In addition, the FPGA's output buffer must be read out by an Arduino microcrontroller and sent to a computer with the right software to parse and display the output. We call these external components macro-components and we call the FPGA internal blocks micro-components.

Macro-Component Descriptions

The required components external to the FPGA are as follows:

Audio signal module

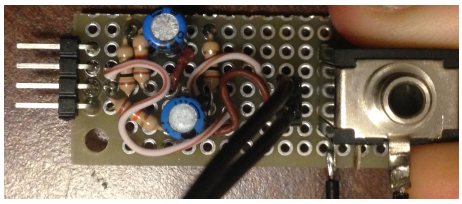

Figure 2: Audio signal circuit board with audio connector

The audio signal module tailors the input audio signal for conversion with the ADC. Traditional audio signals are centered around 0 V and are in the negative voltage range half the time. This module biases the signal by 2.5 V, allowing for a 5 V variation peak to peak in the audio signal without any clipping. This module also provides filtering capacitors that remove most of the supply rail noise that may be present.

ADC PMOD

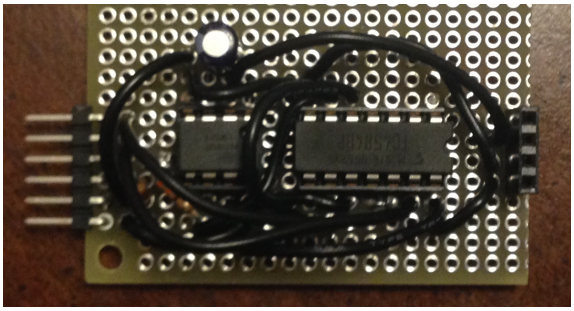

Figure 3: ADC PMOD circuit board

The ADC PMOD accepts any input analog input and converts it to a SPI bus that the FPGA can read. The analog signal is fed into the onboard ADS7818 chip. Unfortunately, this chip operates at 5 V but the FPGA operates at 3.3 V. Because of this, there is a voltage divider to drop the voltage down to the FPGA. For FPGA output to be read by the ADS7818, we added a Schmitt inverter IC wired as a Schmitt trigger to act as a level converter.

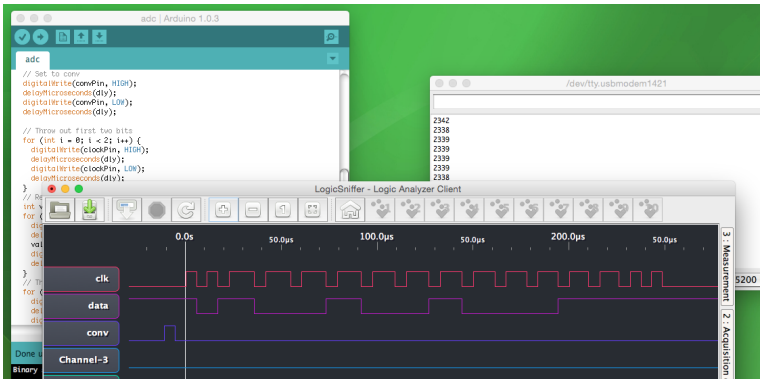

Figure 4: ADC interfacing

We started interfacing with the ADS7818 using an Arduino. Once we understood the protocol completely, we moved the interfacing code from the Arduino to the FPGA and started using real signal data.

Arduino

The Arduino functions as a means of getting the processed information out of the FPGA's BRAM. The Arduino turned out to be the main bottleneck in terms of framerate to the computer's display. We had to push the SPI and serial write functions outside of recommended bounds to get a fluid framerate on the computer display. The full software is available in the `arduino/osc_binary_spi` directory of the source code repository.

Computer software

The computer software for this project is divided into two components. The first component is the Python module that facilitates the reading and decoding of the data from the serial port. It will read out 512 complex numbers and, optionally, compute the magnitude of each of those numbers. The second component is the actual display. It will use the Python module to read the device and it uses Tkinter to display the plot on the screen.

Micro-Component Descriptions

The final block diagram of our device is as follows:

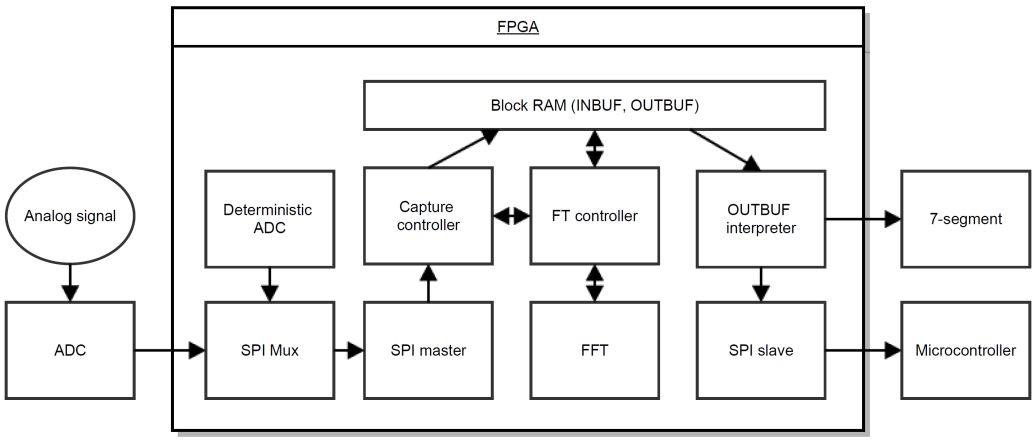

Figure 5: Block diagram of the system and subcomponents

All of the blocks inside the FPGA device are referred to as micro-components.

- Deterministic ADC: Provides a completely predictable ADC signal for testing; it substituted for a real ADC until one was acquired.
- SPI Mux: A simple multiplexer that selects between the external ADC and the deterministic ADC.
- SPI Master: An OpenCore that implements an SPI master interface. This handles the control and reading of the SPI bus from either the deterministic ADC or the external ADC.
- Capture Controller: This block reads samples from the SPI Master and stores the readings as complex numbers in the input buffer. The sample size is 512. It also communicates with the FT controller about when it finishes.
- FT controller: This block cooridinates access to the BRAM between the Capture Controller, FFT Block and OUTBUF interpreter using a simple state machine.

- FFT Block: Performs the FFT when given the start signal from the FT controller. This was instantiated from the Xilinx IPCore Generator library and requires 2 block RAMs and 3 Multipliers, which the Spartan-3E was barely able to accomidate.
- OUTBUF interpreter: This block facilitates the reading of the OUTBUF by feeding words into the SPI slave module.
- SPI Slave: An OpenCore that implements an SPI slave interface. This handles the control and writing of the SPI bus, and it connects to an external microcontroller.

Simulation Results

After the subcomponents of the system were verified individually, they were all assembled together under the FT Controller block, which acted as our top module. The most important shared resource in our design was a single 18k BRAM instantiated inside the FT Controller. The RAM was split into two halves, the first acting as an input buffer and the second acting as an output buffer, and was controlled using four main states:

1. cc_write: The Capture Controller has access to the RAM and writes its data directly to the input buffer.
2. fft_read: The FFT Block reads data from the input buffer. It remains in this state until the FFT calculation is actually performed.
3. fft_write: The FFT Block writes data to the output buffer.
4. oi_read: The OUTBUF Interpreter reads from the output buffer and sends the data out into the arduino.

These steps are then repeated as long as the system is enabled.

Figure 6 shows the wait_for_valid state, which only lasts for several clock cycles. It is an indication from the FFT block that the transform has finished computing and it will be writing data to its output as soon as a valid signal is given. Once valid goes high, the data is written out on the xk_re and xk_im lines. Note that in this simulation, the input was a sawtooth wave, which is

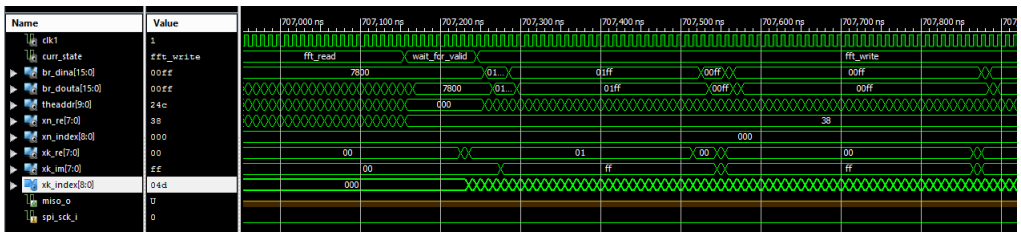

Figure 6: Close up of the transition between fft read and write stages

an odd signal. Thus, ideally all the real parts of the signal would be 0. This is not exactly the case, but it is reasonable, as the real outputs are *mostly* zeros.

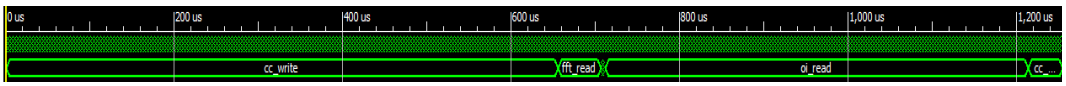

Figure 7: Simulation of the entire processing cycle

The important thing to note in the above figure is the relative lengths of each stage. The actual FFT computation is performed in the very middle of the simulation, and is always approximately 55 us. In practice, the capture controller receives 512 samples over the course of about 6.4 ms, which leaves plenty of time in between cycles to compute the FFT and write the data out to the Arduino. It is smaller here because the cycle would take a very long time to simulate using the real-time capture speed.

Several videos have been uploaded to youtube demonstrating the current capabilities of the system. They are included in the appendix section and can be accessed from the project repository on github as well. Also source code can also be found on github.

Roadblocks

At the start of this project, we planned on computing the DFT entirely out of custom components. However, as the project wore on, we realized that this model was impractical and couldn't be accomplished in the time alotted by two undergraduates with relatively little signal processing and VHDL

experience. Thus, we turned to the Xilinx CORE Generator, which luckily enough had an FFT module available for the Spartan-3E. It utilizes the Cooley-Tukey algorithm as described above, but in a much more professional manner. It also preloads the board with any needed twiddle factors, which was one of the biggest concerns regarding the custom module. In order to change anything about your sample or word sizes, much more code would have to be rewritten, whereas by using the CoreGen model, we can just rebuild the core. This allows for much more flexability in our design, which is very important for a project with so many variables.

Because all of the microcomponents needed to fit on a single Spartan-3E FPGA, some serious size considerations had to be taken into account. First and foremost, there is a very limited pool of resources available on the board. The Spartan-3E only has four BRAMs and four MULTs available. At first we planned on using the Radix-2, Burst I/O implementation of the FFT block, which only used three of each of the MULTs and BRAMs. This synthesized and simulated well, however the tool refused to route the design. This was because of conflicting resources between the BRAMs and MULTs, as they share several internal busses that couldn't be switched fast enough.

In order to remedy the design, the core has another option of Radix-2 Lite, Burst I/O. This uses one less BRAM, but takes about 20 us longer to perform the computation. As mentioned before, this was not a real concern, as we were taking samples as such large intervals that forcing a calculation to last 55 us instead of 35 us was not that big of a deal.

Future Improvements

The largest detractor from our system's functionality is the lack of storage space on the FPGA. Because all of our data needed to be stored as complex numbers, each word was split into a real and complex part, each with a bit width of 8, represented as fixed point 2's complement. In order to store the desired 512 samples, we needed at least 8k RAM, which is close to half of a single BRAM. The other half could then be used to store the processed data to be read by the output interpreter. An 8 bit word does not provide good resolution at all, but increasing it would be unroutable, as another BRAM

would need to be instantiated bringing our total up to four.

If we had more time, there are some improvements that could be made. One is by doubling the word size to 32 and halving the sample size, which wouldn't affect the BRAM configuration at all. This would get rid of many of the uppermost frequencies, but would allow for more accuracy when analyzing the data. Another observation is that the input buffer and output buffer are never accessed at the same time. Thus, they could essentially be merged into a single block without any consequence. The input data would just be overwritten by the output data, which wouldn't be replaced until the capture controller is started up again.

This could allow for the increase from 32 bit to 64 bit words, which is actually large enough to start using IEEE 754 standard floating point arithmetic. Ideally this option would have been implemented. It allows for the most accurate results at the expense of latency, which as we've explained isn't really an issue for this project. Most high end DSPs use floating point calculations as opposed to fixed point, which is much more appropriate considering the amount of mathematical calculations performed.

To combat the growth issues of fixed point multiplication, the FFT block requires a scaling factor field that is proportional to the sample size. After some experimentation, we settled on an adjustment of [2130], which represents the number of bits by which to shift the output data at each even stage of the transform. This is not an ideal solution, as bit shifting already small numbers may just end up ruining what little valid data is present. However, that is one of the compromises of using such a small word size.

Conclusion

In the end, our system was successful, although not nearly as high quality as we were hoping. With more experience, expertise and time however, some great improvements could be made. By implementing a floating point unit and increasing the resolution size, the processed data would be much more accurate. As a whole, this system design was a fight against too few resources. The Spartan-3E FPGA was not designed to be a rigourous calculating ma-

chine; instead it is a simple introduction to the capabilities of FPGAs in general. This is not a bad thing, but its natural limitations start to become an issue when it's being used to fill a need it simply wasn't designed for.

Appendix

1. **Github Repo:** <https://github.com/gmassman/vhdl_fft.git>
2. **Signal Generator Video:** <https://www.youtube.com/watch?v=2sd1Fj6ndW4>
3. **Audio Signal Video:** <https://www.youtube.com/watch?v=x32IYwRD3P8>
4. **Full System Video:** <https://www.youtube.com/watch?v=RKyCnKK4a5A>

References

[1] Doin, Jonny. *SPI Master/Slave Interface.* OpenCores, 16 May 2011. Web. 13 Sept. 2014. <`http://opencores.org/project,spi_master_slave`>.

[2] Reynwar, Ben. *FFT on an FPGA.* FFT on an FPGA. N.p., n.d. Web. 13 Sept. 2014. <`http://www.reynwar.net/ben/docs/fft_dit/index.html`>.

[3] Roberts, Michael J. *Signals and Systems: Analysis Using Transform Methods and MATLAB.* New York: McGraw Hill, 2012. Print.

[4] Satoh, Keiichi, Jubee Tada, Kenta Yamaguchi, and Yasutaka Tamura. *Complex Multiplier Suited for FPGA Structure.* Computers and Communications (2008): 341-44. Web. 13 Sept. 2014.

[5] Wikipedia contributors. *Cooley–Tukey FFT algorithm.* Wikipedia, The Free Encyclopedia. Wikipedia, The Free Encyclopedia, 27 Jun. 2014. Web. 13 Sep. 2014.

[6] Wikipedia contributors. *Discrete Fourier transform.* Wikipedia, The Free Encyclopedia. Wikipedia, The Free Encyclopedia, 2 Sep. 2014. Web. 13 Sep. 2014.

[7] Xilinx. *LogiCORE IP Fast Fourier Transform v7.1* Product Specifications, 1 March 2011.