FPGA implementation of Cooley-Tukey

Garrett Massman and Cory Walker

September 13, 2014

Overview

In this project we will implement the Cooley-Tukey Fast Fourier Transform algorithm on a Xilinx FPGA using VHDL. The primary motivation for this is to digitally analyze musical signals, regardless of the instruments they come from. Using the Cooley-Tukey algorithm, we can calculate the FFT in $O(nlog_2(n))$ time. As far as implementation, the project will be split into several main components. First, we will need to sample our audio for a set amount of time and store the data into a buffer in memory. Then, the digital signal will need to be processed using a complex ALU. Finally, the data will need to be sent to an output buffer for application specific usage. This can be in the form of a 7-segment hex display (for instance if we want to print out the fundamental frequency), or a microcontroller (if we want to plot the entire sampled spectrum by passing it to a more capable platform such as Matlab).

Theory

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

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

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 computer resources. Thus, calculating the DFT in this way is very inefficient, and we must instead turn to the Fast Fourier Transform.

In order to perform this operation more quickly, we can utilize the Cooley-Tukey FFT algorithm. However, the 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

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

In Mathematica

We implemented a candidate version of the Cooley-Tukey algorithm in Mathematica. This code shows how it is possible to build a look up table of calculated results against the variables N and k for the calculation $e^{-\frac{2\pi i}{N}k}$. This table will not only save us time, but also complexity in the design by allowing us to simply look up this value.

```
myfasterft[x_, capn_, s_, precomputed_] := (
   result = {};
   If[capn == 1,
    result = Append[result, x[[1]]],
    result = Join[result, myfasterft[x, capn/2, 2 s, precomputed]];
    result =
     Join[result, myfasterft[x[[s + 1 ;;]], capn/2, 2 s,
        precomputed]];
    Do[
     t = result[[k + 1]];
     result[[k + 1]] =
      t + precomputed[[Log2[capn]]][[k + 1]]*
       result[[k + capn/2 + 1]];
     result[[k + capn/2 + 1]] =
      t - precomputed[[Log2[capn]]][[k + 1]]*
       result[[k + capn/2 + 1]];
     , {k, 0, capn/2 - 1}];
    ];
   result
   );

precomputedmatrix[datalen_] := (
  Table[
   Table[
    E^(-I*2*\[Pi]*k/capn),
    {k, 0., datalen/2 - 1}
    ],
   {capn, Table[2^x, {x, 1, Log2[datalen]}]}
   ]
  )

data = Table[N[Sin[30 2 Pi n/200] + (RandomReal[] - 1/2)], {n,
```

```
    512}];
precomputed = precomputedmatrix[Length[data]];
ListLinePlot[Abs[myfasterft[data, Length[data], 1, precomputed]],
    PlotRange -> All]
```

FPGA considerations

Because we are working with a very specialized type of hardware, we must take into consideration some limiting parameters. In particular, the Spartan3Es loaded on each BASYS board only have 72K RAM. This means that if we want a real time FFT, we will need to optimize the space we have available, and only write to it when it is absolutely necessary. We must also consider the quality of our digital input signal. There are various ADC PMODs available for the BASYS boards, which range in quality from 4.8 KHz to 1 MHz in speed. If we are only sampling audio, we probably won't need to gather any frequencies above 1200 Hz, so the lower end model should be fine.

Once our data has been collected and stored inside the FPGA, it needs to be processed. This is perhaps the most challenging part of the project, as we need to implement what will be called an FTC (Fourier Transform Controller) as well as a CALU (Complex Arithmetic Logic Unit). The FTC's primary function will be to keep track of the data as it moves from RAM, into the CALU, and back out into the real world. This will require some heavily synchronized logic and very tight timing control to be carried out properly. It will work in sync with the SPI input master to tell the CALU when data is ready to be read, and then clear the input buffer once the CALU has finished. It will also need to monitor the movement of data from the output buffer into whatever output stream requires it. The data may be sent directly to a microcontroller or into another component for post processing.

The CALU will have one primary job, which is to actually perform the FFT. From its perspective, it will only the addresses of an input buffer and an output buffer, as well as a flag that tells it when to start and stop. Because all of our calculations are being done inside of an FPGA, the CALU will be most composed of combinational logic at its core. In theory (i.e., provided enough gates), all that needs to be built is the base case of the FFT, which

takes in two numbers and outputs their sum and difference.

$$y_0 = x_0 + e^{-\frac{2\pi i}{N}k}x_1 \qquad\qquad y_1 = x_0 - e^{-\frac{2\pi i}{N}k}x_1$$

This portion of the computation is commonly called a *butterfly*, because the system's diagram looks vaguely like a butterfly. Each of the twiddle factors included above can be calculated beforehand and stored in a lookup table for quick access before each iteration. Because we will be performing the same type of computation with them each time, they are essentially constants in our system.

Block diagram

Over the past few weeks we have taken what we know and outlined a basic block diagram of our device:

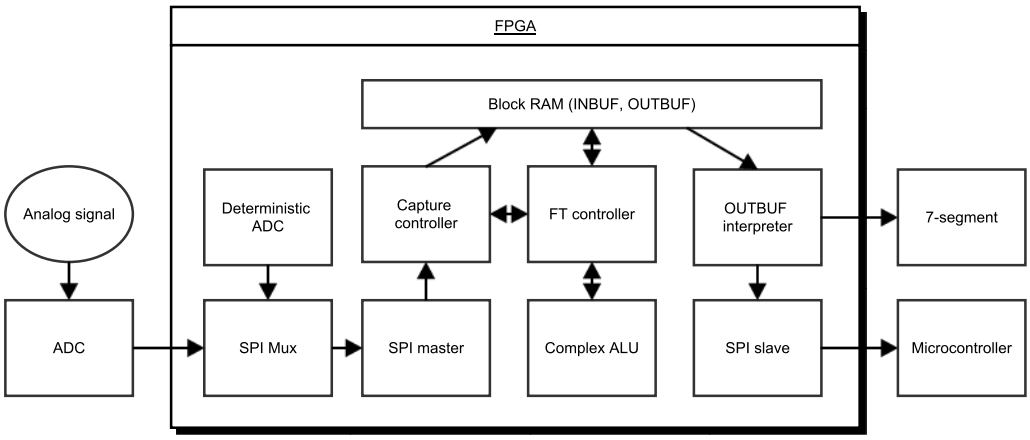

Figure 1: Block diagram of the FFT chip

Now that we have divided the device into several logical blocks, we can now split up the work accordingly.

Roadmap

In this section, we will describe the steps, in order, that need to be done. Our steps will essentially follow the data pipeline of the module:

- We will begin by implementing a deterministic "ADC" module that will emulate the interface of the real ADC but generate a fixed signal every time. We will use this signal in our own calculations so that we can then verify the outputs of the chip at each stage in the data pipeline. When we are ready to convert to real ADC signals, we can simply switch to an external ADC module using an SPI mux. This step has already been completed.

- We will then implement the capture controller. This controller will have an SPI master unit that can read from the ADC. It will be also connected to the block RAM to write the input buffer.

- Next, we will work on the heart of the chip, the FT controller. This controller will first simply implement the DFT algorithm. It will utilize the complex number ALU to make calculations.

- After the FT controller, we will implement an OUTBUF interpreter that is responsible for outputting the results of the FT controller to the rest of the world using SPI.

- Once all of this is done, we will finish converting our FT controller from using the simple DFT algorithm to the more complex Cooley-Tukey FFT algorithm.

- If we have further time still, we will build interesting demonstrations of this device.

Resources

While our aim is to implement the entire Fast Fourier Transform with our own code, there are some blocks of our design that are common practice to reuse. Those blocks are the SPI modules. While SPI is not a complicated communication protocol, it requires a fair amount of time to implement. There is a great resource online called OpenCores. This resource is essentially a collection of open source HDL modules that can be freely used in designs.

We will be using the "SPI Master/Slave Interface" module a few times in our design to implement SPI.

Possible extensions

The Fast Fourier Transform algorithm applied to an arbitrary voltage signal is very useful. Because our final chip will provide such capability, we have the option of using our newly-designed chip with multiple different applications.

One of our ideas is to attach a microphone and a preamp and use the frequency-domain signal to determine the musical note being played. We could also build a tuner for a guitar.

Additionally, we could connect an audio signal directly from any music player and display a graphic equalizer effect using either a computer or microcontroller that will interface with our chip.

Finally, provided we are able to tune our chip and ADC to operate fast enough, we might be able to build a radio wave analyzer for very long wavelength signals. This device would almost certainly be too slow for any serious RF work, but it would still be an interesting extension.

These extensions are not crucial to our design but might serve as a great demonstration when we present the device to the class.

Conclusion

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.