Gisselquist
Technology, LLC

PIPELINED FFT SPECIFICATION

Dan Gisselquist, Ph.D.
dgisselq (at) opencores.org

January 23, 2021

Revision History

Rev.	Date	Author	Description
0.3	6/2/2015	Gisselquist	General purpose pipelined FFT generator
0.2	6/2/2015	Gisselquist	Superficial formatting changes
0.1	3/3/2015	Gisselquist	First Draft

Contents

Figures

Figure Page

Tables

Table Page

Preface

This FFT came about originally from my attempts to design and implement a GPS decorrelation algorithm inside a generic FPGA, but only on a limited budget. As such, it was built before I had the board that could use it. Because I was trying to hit a very high processing rate, the FFT core was originally built to handle two samples at a time. Hence the original name, the "Double clocked FFT core".

One of the difficulties of doing all of your development using Verilator as a simulator, is that you can only simulate components that you have the Verilog source for. My desire to use Verilator kept me from using any of the vendor supplied FFTs out there.

This, then, was and is the genesis of this project.

Since this genesis, I've used this core as part of several other designs and maintaied it. Eventually it has morphed into the general purpose FFT core generator that it is today.

Dan Gisselquist, Ph.D.

1.

Introduction

The General Purpose Pipelined FFT generator project contains all of the software necessary to create an arbitrary sized FFT HDL core that will accept up to two samples per clock cycle, and after some pipeline delay it will output FFT results at the same rate they are input.

The FFT generated by this approach is very configurable. By simple adjustment of a command line parameter, the FFT may be made to be a forward FFT or an inverse FFT. The number of bits processed, kept, and maintained by this FFT are also configurable. Even the number of bits used for the twiddle factors, or whether or not to bit reverse the outputs, are all configurable parts to this FFT core. Finally, the FFT can be configured to process two samples per clock, one sample per clock, one sample every other clock, or even one sample every third clock (or less).

These features make this general purpose pipelined FFT generator very different and unique among the other cores available on opencores.com.

For those who wish to get started right away, please download the package, change into the `sw` directory and run `make`. There is no need to run a configure script, `fftgen` is completely portable C++. (While I do my development on Ubuntu, I am told by others that the core builds on Microsoft systems as well.) Then, once built, go ahead and run `fftgen` without any arguments. This will cause `fftgen` to print a usage statement to the screen. Review the usage statement, and run `fftgen` a second time with the arguments you need.

2.

Generation

Creating an FFT core is as simple as running the program `fftgen`. The program will then create a series of Verilog files, as well as `.hex` files suitable for use with a $readmemh, and place them into an output directory, `./fft-core/` by default, that `fftgen` will create. Creating the core you want takes a touch of configuring. Therefore, the following lists the arguments that can be given to `fftgen` to adjust the core that it builds:

-f size This specifies the size of the FFT core that `fftgen` will build. The size must be a power of two.

Given an input $x[n]$, the FFT will calculate,

$$X[k] = \sum_{n=0}^{N-1} x[n]\, e^{-j2\pi \frac{k}{N} n}$$

to within a scale factor.

-i This specifies that the FFT will be an inverse FFT. Specifically, it will calculate,

$$x[n] = \sum_{k=0}^{N-1} X[k]\, e^{j2\pi \frac{k}{N} n}$$

If no `-i` option is given, the core will by default generate a forward FFT.

-2 Builds an FFT that can ingest and output two samples per clock.

This option requires six multiplies for all but the last two butterfly stages. The last two butterfly stages are accomplished using shifts and adds only, so they require no multiplies.

-k 1 Builds an FFT that can ingest and output one sample per clock. This option is incompatible with `-2`.

This option requires three multiplies for all but the last two butterfly stages.

-k 2 Builds an FFT that can ingest and output one sample every other clock. This option is incompatible with `-2`, and will override `-k 1`. You are responsible for making sure that `i_ce` will never be true for two clocks in a row.

Unlike `-k 1`, this option only requires two multiplies for all but the last two butterfly stages.

-k 3 Builds an FFT that can ingest and output one sample every third clock. This option is incompatible with `-2`, and will override any other `-k` option. For this to work, you will need to guarantee that `i_ce` will never be true more than one time in any three clock periods.

Unlike `-k 1` and `-k 2`, this option only requires one multiply for all but the last two butterfly stages.

-s This causes the core to skip the final bit reversal stage. The outputs of the FFT will then come out in bit reversed order.

This option is useful in those cases where someone wishes to multiply the coefficients coming out of an FFT by some product, and then to inverse FFT the results. If the coefficients are also applied in bit–reversed order, then both the FFT and IFFT may skip their bit reversals.

Be aware, however, doing this requires the bit reversed forward transform be followed by a bitreversed decimation in time approach to the inverse transform. This software does not (yet) provide that capability. As such, this capability is really just a placeholder for a future capability.

-d DIR Specifies the DIRectory to place the produced Verilog files. By default, this will be in the './fft-core/' directory, but it can be moved to any other directory as necessary.

-n bits Sets the number of input bits per sample. Given this setting, each of the two samples clocked in at every clock cycle will have this many bits for their real portion, and again this many bits for their imaginary portion. Thus, the data input to the FFT will be four times this many bits per clock.

-m bits This sets the maximum bit width of the output. By default, the FFT will gain bits as they accumulate within the FFT. Bits are accumulated at roughly one bit for every two stages. However, if this value is set, bits are only accumulated up to this maximum width. After this width, further accumulations are truncated.

-c bits Specifies the number of extra bits to be given to each twiddle factor. The size of the twiddle factors is nominally the size of the input data. By specifying `-c <bits>`, you can extend this default value to avoid any loss in precision.

-x bits Maintains `bits` extra bits during the computation to deal with roundoff error. Hence, after the first stage, there will be this many excess bits within the FFT pipeline.

Internally accumulated roundoff error can be a difficult problem to solve. By using this option, you guarantee that the FFT runs with an additional `bits` bits, and only truncates down to the necessary width at the end in order to minimize rounding errors along the way.

-p nmpy This sets the number of hardware multiplies that the FFT will consume. By default, the FFT does not use any hardware multiplies. However, this can be expensive on the rest of the logic used by the device. You can avoid this problem by allowing the FFT to use hardware multiplies using this option. By default, the multiplies will be used in the latter stages, so that they will be applied where the bit width is the greatest.

3.

Architecture

As a component of another system the structure of this system is a simple black box such as the one shown in Fig. 3.1 for the two traditional one sample per clock FFT implementation, or Fig. 3.2 for the two samples per clock FFT.

The interface is simple: strobe the reset line, and every clock thereafter set the clock enable line when data is valid on the input port. Likewise for the outputs, when the `o_sync` line goes high the first data sample is available. Ever after that, one data sample will be available every clock cycle that the `i_ce` line is high.

Internal to the FFT, things are a touch more complex.

Fig. 3.3 attempts to show some of this structure for the two-sample per clock FFT. As you can see from the figure, the FFT itself is composed of a series of stages. For the two-sample per clock FFT, these stages are split from the beginning into an even stage and an odd stage. Further, they are numbered according to the size of the FFT they represent. Therefore the first stage is numbered N and represents the first stage of an N point FFT. The second stage is labeled $N/2$, then $N/4$, and so on down to $N = 8$. The four sample stage and the two sample stages are different, however. These two stages, representing three blocks on Fig. 3.3, can be accomplished without any multiplies. Therefore they have been accomplished separately. Likewise all of the stages, save the double stage at the bottom, operate on one data sample per clock. Only the last stage, prior to the bit reversal stage, takes two data samples per clock as input, and outputs two data samples per clock. Finally, the bit reversal stage acts as the last piece of the structure.

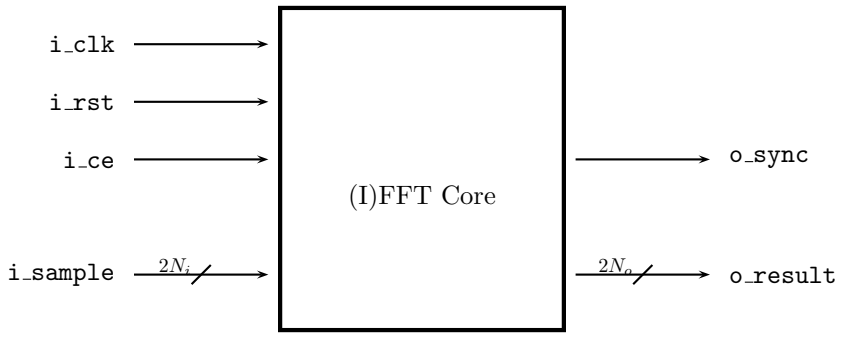

Figure 3.1: (I)FFT Black Box Diagram

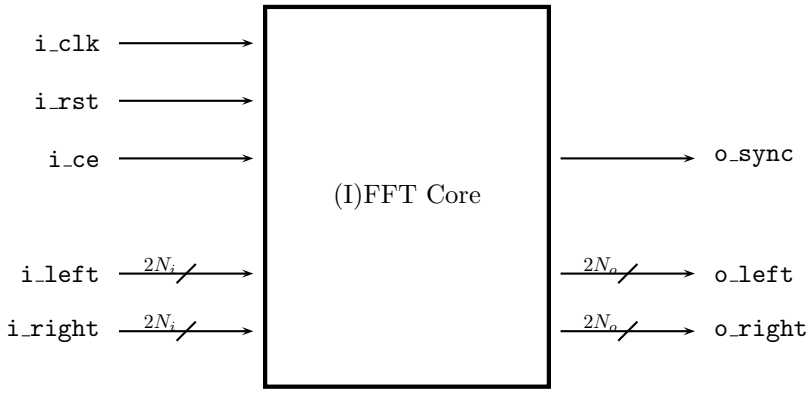

Figure 3.2: Two sample per clock (I)FFT Black Box Diagram

Internal to each of the FFT stages is a butterfly and a complex multiply, as shown in Fig. 3.4. These FFT stages are really no different than any other decimation in frequency radix-2 FFT implementation, save only that for the two-sample per clock FFT the coefficients are alternated between the two inputs shown in Fig. 3.3. That is, the even stages would get all the even coefficients, and the odd stages get all of the odd coefficients. For the more general purpose FFT, there's only the one pipeline, so every sample goes through every butterfly. Internally, each stage spends the first $N/2$ clocks storing its inputs into memory, and then the next $N/2$ clocks pairing a stored input with a single (fresh) external input, so that both values become inputs to the butterfly. Likewise, the butterfly coefficient is read from a small ROM table.

One trick to making the FFT stage work successfully is synchronization. Since the shift and add multiplies create a delay of (roughly) one clock cycle per two bits of input, there is a significant pipeline delay from the input to the output of the butterfly routine. To match this delay, the FFT stage places a synchronization pulse into the butterfly. When this synchronization pulse comes out of the butterfly, the values of the butterfly then match the first sample out of the stage. The next synchronization problem comes from the fact that the butterflies operate on two samples at a time, whereas the FFT stage operates on a single sample at a time. This means that half the time the butterfly output will be invalid. To keep things aligned, and to avoid the invalid data half, a counter is started by the synchronization pulse coming out of the butterfly in order to keep track. Using this counter and once the butterfly produces the first sync pulse, the next $N/2$ clock cycles will produce valid butterfly outputs. For these clock cycles, the left or first output is sent immediately to the next FFT stage, whereas the right or second output is saved into memory. Once these cycles are complete, the butterfly outputs will be invalid for the next $N/2$ clock cycles. During these invalid clock cycles, the FFT stage outputs data that had been stored in memory. In this fashion, data is always valid coming out of each FFT stage once the initial synchronization pulse goes high.

The complex multiply itself, formed internal to the butterfly routine, is formed from three very simple shift and add multiplies, whose output is then transformed into a single complex output, although there is a command line option to use hardware multiplies instead. To avoid overflow, the

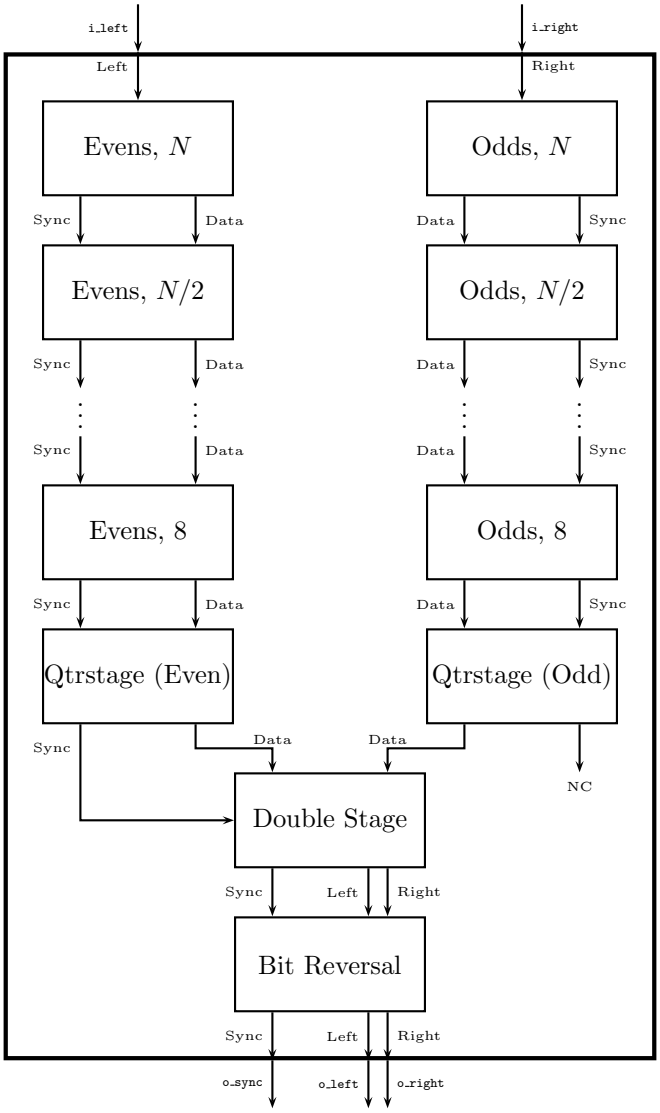

Figure 3.3: Internal FFT Structure

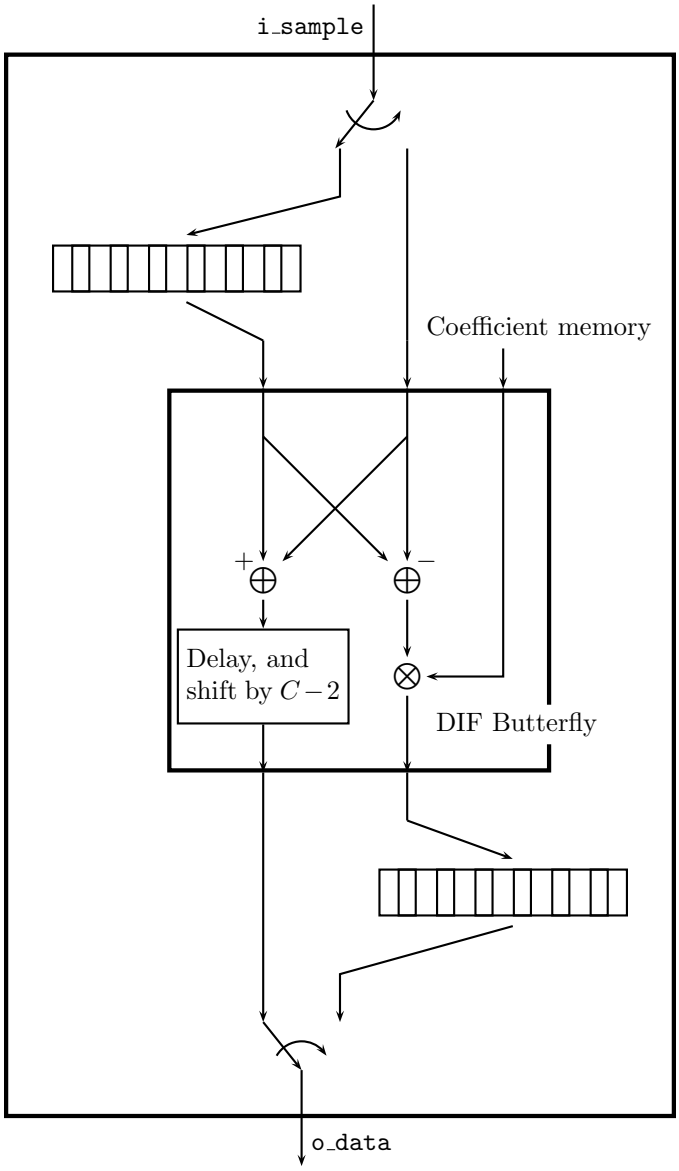

Figure 3.4: A Single FFT Stage, with Butterfly

complex coefficients, z_n, for these multiplies are given by,

$$z_n = c_n + js_n, \text{ where} \tag{3.1}$$

$$c_n = \left\lfloor 2^{C-2} \cos\left(2\pi \frac{n}{N}\right) + \frac{1}{2} \right\rfloor, \tag{3.2}$$

$$s_n = \left\lfloor 2^{C-2} \sin\left(2\pi \frac{n}{N}\right) + \frac{1}{2} \right\rfloor, \text{ and} \tag{3.3}$$

C is the number of bits allocated to the coefficient.

For those wishing to understand this operation further and in more depth, I would commend them to the literature on how a decimation in frequency FFT is constructed.

4.

Operation

The core is actually really easy to use:

1. Provide a system clock to the core every clock cycle.
2. Set the `i_rst` line high for at least one clock cycle before you intend to use the core.
3. From the time of reset until the first sample pair is available on the IO ports, `i_rst` may be kept low, but the clock enable line `i_ce` must also be kept low.
4. On the clock containing the first sample, `i_sample`, or the first sample pair, `i_left` and `i_right` for the two sample-per-clock FFT, set `i_ce` high.

   If you have elected an FFT that multiplexes its multiplies, and so can only handle one sample every two or three clocks, then you'll need to guarantee `i_ce` remains low for one or two clocks respectively before raising it again.
5. Ever after, any time a valid sample is placed in `i_sample` and `i_ce` raised high, a sample will enter into the FFT for processing. For the two sample-per-clock FFT, the input will be into `i_left` (for the first input) and `i_right` (for the next one).
6. At the first valid output, the FFT core will set `o_sync` line high in addition to placing the output value into `o_result`. For the two-sample per clock FFT, the outputs will be placed into `o_left` (the first of two), and `o_right` (the second of the two) respectively.
7. Ever after, whenever `i_ce` is high, the FFT core will clock two samples in and two samples out. On any valid first pair of samples coming out of the transform, `o_sync` will be high. Otherwise `o_sync` will remain low.

There are no special modes or states associated with this core. If you wish it to stop or pause, just turn off `i_ce`. If you wish to flush the core, just send zeros into the core.

5.

Registers

Once built, the FFT routine has no capability for runtime configuration or reconfiguration. Therefore, this implementation maintains no user configurable or readable registers.

This is a great advantage in many ways, simply because it greatly simplifies the interface over other cores that are available out there.

6.

Clocks

The FFT routines built by this core use one clock only. The speed of this clock will depend upon the speed your hardware is capable of. If your data rate is slower than your clock speed, just hold off on the `i_ce` line as necessary so that every clock with the `i_ce` line high is a valid sample.

7.

IO Ports

The FFT core presents a small set of IO ports to its external interface. These ports are listed in Table. 7.1.

Port	Width	Direction	Description
i_clk	1	Input	The global clock driving the FFT.
i_rst	1	Input	An active high synchronous reset.
i_ce	1	Input	Clock Enable. Set this high to clock data in and out.
i_sample	$2N_i$	Input	The complex input sample. Bits $[(2N_i-1):N_i]$ of this value are the real portion, whereas bits $[(N_i-1):0]$ represent the imaginary portion. Both portions are in signed twos complement integer format. The number of bits, N_i, is configurable.
i_left	$2N_i$	Input	When the core is configured for two-samples per clock, this is the first of the two data inputs presented to the core on any given clock. It has the same format as `i_sample` above.
i_right	$2N_i$	Input	The second of two input complex input samples, used when the core is configured for two-samples per clock. The format is the same as `i_sample` above.
o_sync	1	Output	Signals the first output sample of any transform. It will be zero from the time of the reset until the first output sample. Ever afterwards, it will be true any time bin zero of the FFT is on the output.
o_result	$2N_o$	Output	The complex output sample. The format is the same, save only that N_o bits are used for each twos complement portion instead of N_i. Hence bits $[(2N_o-1):N_o]$ of this value are the real portion, whereas bits $[(N_o-1):0]$ represent the imaginary portion.
o_left	$2N_o$	Output	When in the two-sample per clock configuration, this is the first of two complex output samples.
o_right	$2N_o$	Output	When in the two-sample per clcok configuration, this is the second of two complex output samples.

Table 7.1: List of IO ports