The Fast Fourier Transform in Hardware: A Tutorial Based on an FPGA Implementation

Article · March 2013

CITATION
1

READS
12,422

1 author:

George Slade
Orban Microwave Products
44 PUBLICATIONS **130** CITATIONS

SEE PROFILE

The Fast Fourier Transform in Hardware: A Tutorial Based on an FPGA Implementation

G. William Slade

Abstract

In digital signal processing (DSP), the fast fourier transform (FFT) is one of the most fundamental and useful system building block available to the designer. Whereas the software version of the FFT is readily implemented, the FFT in hardware (i.e. in digital logic, field programmable gate arrays, etc.) is useful for high-speed real-time processing, but is somewhat less straightforward in its implementation. The software version is generally constrained to execute instructions serially (one at a time) and is therefore severely constrained by the processor instruction throughput. The hardware FFT performs many of its processing tasks in parallel, hence can achieve order-of-magnitude throughput improvements over software FFTs executed in DSP microprocessors. However straightforward the FFT algorithm, when implementing the FFT in hardware, one needs to make use of a number of not-so-obvious tricks to keep the size and speed of the logic on a useful, practical scale. We do not present this document as an exhaustive study of the hardware fourier transform. On the other hand, we hope thet reader comes away with an understanding on how to construct a basic, but useful FFT calculator that can be the basis for deeper study as well as future improvements and optimization.

In this article, we focus on the Cooley-Tukey Radix-2 FFT algorithm [6], which is highly efficient, is the easiest to implement and is widely used in practice. We review the mathematical basis of the algorithm and its software implementation before launching into the description of the various system blocks needed to implement the hardware version of the FFT. We then describe how the FFT is instantiated in a field programmable gate array (FPGA) and used in a real system. It is hoped that by reading this document, the reader will have a good grasp on how to implement a hardware FFT of any power-of-two size and can add his own custom improvements and modifications.

I. Introduction

A. The DFT: Discrete Fourier Transform

The DFT is a linear transformation of the vector x_n (the time domain signal samples) to the vector X_m (the set of coefficients of component sinusoids of time domain signal) using

$$X_m = \sum_{n=0}^{N-1} x_n w^{nm}, \tag{1}$$

where N is the size of the vectors, $w = e^{2i\pi/N}$ are the "roots-of-unity" (twiddle factors), and $0 \leq m < N$. A brute-force summation requires on the order of N^2 operations to compute. This rapidly becomes intractible as the number of samples becomes large. A very useful strategy is to recursively split the summation like this:

$$X_m = \sum_{n=0}^{N/2-1} x_n w^{nm} + w^{mN/2} \sum_{n=0}^{N/2-1} x_{n+N/2} w^{nm}, \tag{2}$$

Author can be reached at *bill.slade@ieee.org*

or, like this:

$$X_m = \sum_{n=0}^{N/2-1} x_{2n} w^{2nm} + w^m \sum_{n=0}^{N/2-1} x_{2n+1} w^{2nm}. \tag{3}$$

We see immediately that in both cases, that any DFT can be constructed from the sum of two smaller DFTs. This implies that we can attack the problem using the "divide and conquer" approach. The summation is applied recursively to ever-smaller groups of sample data providing us with an algorithm whose computational cost is proportional to $N \log_2 N$; a substantial savings in effort! As a result, we must work with vector sizes that are powers-of-two. (In reality, it is not much of a drawback if we "pad" unused samples with zeros.)

We note that there are different ways to partition the summations. We have shown two of the most popular methods in (2) and (3). The expression in (2) represents the so-called decimation-in-frequency (DIF) split, whereas (3) is the decimation in time (DIT) split. It is the DIT form of the FFT that we concentrate on in this paper.

It is worth mentioning that other splits and ordering methods exist. The Winograd algorithm, for example, uses a special ordering to reduce the need for complex multiplications [1], [2]. Other algorithms rely on the Chinese Remainder Theorem (Prime-factor algorithm [4]). The Cyclic Convolution Method [3] can also handle prime or nearly prime vector sizes. Yet another elegant trick for carrying out the Fourier transform if the Chirp-z algorithm [5]. These methods each have their advantages and disadvantages. The mathematical basis of these alternative methods is often very elegant, but the ordering methods are usually not so obvious to the beginner wishing to implement a Fourier transform on his/her FPGA demo board. Moreover, it is difficult to beat the simplicity and speed of the power-of-two divide-and-conquer methods. For this reason, we focus on the Cooley-Tukey method and refer any interested readers to the papers in the list of references.

Let us consider the DFT acting on a vector of size 8 to illustrate how the algorithm is formed. We write out the summations for X_m expanding the powers of w in matrix form:

$$\begin{pmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \\ X_4 \\ X_5 \\ X_6 \\ X_7 \end{pmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & w & w^2 & w^3 & w^4 & w^5 & w^6 & w^7 \\ 1 & w^2 & w^4 & w^6 & w^8 & w^{10} & w^{12} & w^{14} \\ 1 & w^3 & w^6 & w^9 & w^{12} & w^{15} & w^{18} & w^{21} \\ 1 & w^4 & w^8 & w^{12} & w^{16} & w^{20} & w^{24} & w^{28} \\ 1 & w^5 & w^{10} & w^{15} & w^{20} & w^{25} & w^{30} & w^{35} \\ 1 & w^6 & w^{12} & w^{18} & w^{24} & w^{30} & w^{36} & w^{42} \\ 1 & w^7 & w^{14} & w^{21} & w^{28} & w^{35} & w^{42} & w^{49} \end{bmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{pmatrix} \tag{4}$$

Now, let us reorder the matrix according to the DIT split in (3), separating the even and odd index samples,

viz.:

$$\begin{pmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \\ X_4 \\ X_5 \\ X_6 \\ X_7 \end{pmatrix} = \left[\begin{array}{cccc|cccc} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & w^2 & w^4 & w^6 & w & w^3 & w^5 & w^7 \\ 1 & w^4 & w^8 & w^{12} & w^2 & w^6 & w^{10} & w^{14} \\ 1 & w^6 & w^{12} & w^{18} & w^3 & w^9 & w^{15} & w^{21} \\ \hline 1 & w^8 & w^{16} & w^{24} & w^4 & w^{12} & w^{20} & w^{28} \\ 1 & w^{10} & w^{20} & w^{30} & w^5 & w^{15} & w^{25} & w^{35} \\ 1 & w^{12} & w^{24} & w^{36} & w^6 & w^{18} & w^{30} & w^{42} \\ 1 & w^{14} & w^{28} & w^{42} & w^7 & w^{21} & w^{35} & w^{49} \end{array}\right] \begin{pmatrix} x_0 \\ x_2 \\ x_4 \\ x_6 \\ \hline x_1 \\ x_3 \\ x_5 \\ x_7 \end{pmatrix} \tag{5}$$

Let us do the same reordering confined to each 4x4 block to yield:

$$\begin{pmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \\ X_4 \\ X_5 \\ X_6 \\ X_7 \end{pmatrix} = \left[\begin{array}{cc|cc|cc|cc} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & w^4 & w^2 & w^6 & w & w^5 & w^3 & w^7 \\ \hline 1 & w^8 & w^4 & w^{12} & w^2 & w^{10} & w^6 & w^{14} \\ 1 & w^{12} & w^6 & w^{18} & w^3 & w^{15} & w^9 & w^{21} \\ \hline 1 & w^{16} & w^8 & w^{24} & w^4 & w^{20} & w^{12} & w^{28} \\ 1 & w^{20} & w^{10} & w^{30} & w^5 & w^{25} & w^{15} & w^{35} \\ \hline 1 & w^{24} & w^{12} & w^{36} & w^6 & w^{30} & w^{18} & w^{42} \\ 1 & w^{28} & w^{14} & w^{42} & w^7 & w^{35} & w^{21} & w^{49} \end{array}\right] \begin{pmatrix} x_0 \\ x_4 \\ \hline x_2 \\ x_6 \\ \hline x_1 \\ x_5 \\ \hline x_3 \\ x_7 \end{pmatrix} \tag{6}$$

We are now closing in on the point where the FFT "magic" begins to happen. We now invoke the symmetries in the powers of w, the roots-of-unity. Namely,

$$w^n = w^{n+Nk}, N = 8, k = 0, 1, 2, \cdots \tag{7}$$

$$w^n = -w^{n+N/2} \tag{8}$$

$$w^{Nk} = 1. \tag{9}$$

We now rewrite (6) as

$$\begin{pmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \\ X_4 \\ X_5 \\ X_6 \\ X_7 \end{pmatrix} = \left[\begin{array}{cc|cc|cc|cc} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & w^2 & -w^2 & w & -w & w^3 & -w^3 \\ \hline 1 & 1 & -1 & -1 & w^2 & w^2 & -w^2 & -w^2 \\ 1 & -1 & -w^2 & w^2 & w^3 & -w^3 & w & -w \\ \hline 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\ 1 & -1 & w^2 & -w^2 & -w & w & -w^3 & w^3 \\ \hline 1 & 1 & -1 & -1 & -w^2 & -w^2 & w^2 & w^2 \\ 1 & -1 & -w^2 & w^2 & -w^3 & w^3 & -w & w \end{array}\right] \begin{pmatrix} x_0 \\ x_4 \\ \hline x_2 \\ x_6 \\ \hline x_1 \\ x_5 \\ \hline x_3 \\ x_7 \end{pmatrix} \tag{10}$$

After this rearrangement, we notice that we do not need all the powers of w up to 7. We need store only

TABLE I

ILLUSTRATION OF THE BIT-REVERSED INDICES.

Index	binary	Bit reversed index	binary
0	000	0	000
1	001	4	100
2	010	2	010
3	011	6	110
4	100	1	001
5	101	5	101
6	110	3	011
7	111	7	111

those up to 3, because of the sign symmetry of w. Furthermore, the DFT of a two point data set is simply

$$\begin{pmatrix} X_0 \\ X_1 \end{pmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{pmatrix} x_0 \\ x_1 \end{pmatrix}. \tag{11}$$

Taking a close look at the ordering if the x vector, we notice that if we represent the indices as binary numbers, they correspond to the *bit reversed* representation of the original indices. Table I shows how this works for our 8 sample example.

So, now we know that the DIT algorithm consists of a bit-reversal permutation of the input data indices followed by a recursive transformation. The recursive sum in (3) can be represented as a sequence of matrix transformations, viz.:

$$(X) = [A_2][A_1][A_0][P](x), \tag{12}$$

where $[P]$ is matrix representation of the bit-reversal permutation of the original data vector (x). It is easy to see that $[A_0]$ is the first transformation:

$$[A_0] = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 \end{bmatrix} \tag{13}$$

If we use the same atomic two-point transform prototype on each two-by-two matrix and apply the necessary

delay of w^2, we can get the four-by-four transform blocks using $[A_1]$:

$$[A_1] = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & w^2 & 0 & 0 & 0 & 0 \\ 1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & -w^2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & w^2 \\ 0 & 0 & 0 & 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & -w^2 \end{bmatrix} \tag{14}$$

Now, the same technique to the four-by-four blocks to generate the 8 by 8 matrix:

$$[A_2] = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & w & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & w^2 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & w^3 \\ 1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & -w & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & -w^2 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & -w^3 \end{bmatrix}. \tag{15}$$

In fact, at any level l (from 0 to 2 in our present case), we can define a $2^{l+1} \times 2^{l+1}$ matrix template:

$$[T]_l = \begin{bmatrix} [I] & [\Omega] \\ [I] & [-\Omega] \end{bmatrix} \tag{16}$$

where $[\Omega] = \text{diag}(w^0, w^{N/2^{l+1}}, w^{2 \cdot N/2^{l+1}}, w^{3 \cdot N/2^{l+1}}, \cdots)$ and $[I]$ is a $2^l \times 2^l$ identity matrix block. With this template, we can explicitly generate each level of the transform. Interestingly, this approach demonstrates that the FFT is nothing more than a special form of matrix factorization.

Each of the partial transforms corresponds to a level with $2N$ complex multiply-adds. The full transform requires $2N \log_2 N$ multiply-add cycles. The graph in Fig. 1 illustrates the data flow; moving toward each vertex indicating the fetch-multiply-add-store operations. The graph provides us with a processing template. The input data must be in bit-reversed order. Output data will appear in natural order. The full transform requires

1) an address generator,
2) a “butterfly” operator to do the complex multiply/add,
3) a memory and
4) roots-of-unity (twiddle factor) generator.

The address generator provides the locations for the fetch and store operations to and from memory. The butterfly operator is the heart of the FFT. It provides the recursive two-point transforms (the multiply-adds) that are built up to construct the complete transform. The memory is needed to store the intermediate results as the transform runs. The twiddle factor generator can be based on a simple look-up table (used here) or, to save memory, computed on the fly using CORDIC [9].

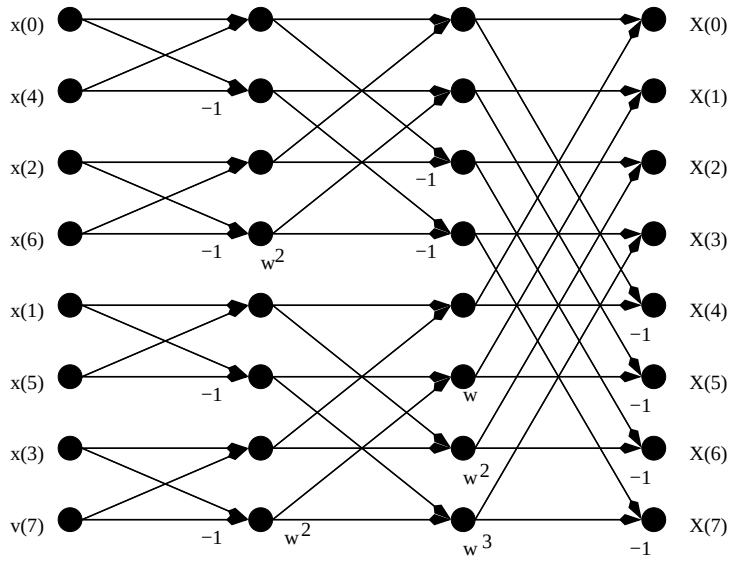

Fig. 1. Signal flow graph for 8 point FFT.

II. Software transform

The software transform is constructed in a straightforward manner by first doing the permutation of the input data and then carrying out the butterfly operations.

Most normal CPUs and high level computer languages have no way to directly perform the bit-swapped reordering of the data, so a fairly cumbersome integer arithmetic sorting trick is used (from [6]) to do the permutation.

Require: $x_n \leftarrow$ data, $N \leftarrow$ # data points ▷ Initialize variables
  procedure PERMUTE(N, **x**)
    $i \leftarrow 1$
    for $n = 1 \rightarrow N$ **do** ▷ Step through each data index.
      if $n > i$ **then**
        $Swap\ x_n \leftrightarrow x_i$ ▷ Use swaps to sort data into bit-reversed address order.
      end if
      $m \leftarrow N/2$
      while $m \geq 2$ && $i > m$ **do** ▷ Compute the new offset for the swap.
        $i \leftarrow i - m$
        $m \leftarrow m/2$
      end while
      $i \leftarrow i + m$
    end for
  end procedure

Many digital signal processors simplify the reordering by either having an explicit bit-reversal instruction or

a bit-reversed addressing mode that is specifically included to facilitate the construction of FFT algorithms. Of course, a hardware FFT constructed in an FPGA easily permits bit reversed addressing by just reversing the physical connections of the data address bus lines.

After the input data is properly ordered, the butterfly operations are executed on pairs of data samples, stepping sequentially through each of the $\log_2 N$ levels. This is the Danielson-Lanczos algorithm [6]. The twiddle-factors w_m contain half of the N "roots of unity".

procedure DANIELSONLANCZOS(**x**, **w**, N)
  $M = 1$ ▷ Set first level of "butterflies."
  while $N > M$ **do**
    $Istep \leftarrow M \ll 1$ ▷ The "stride" of the butterfly computations.
    for $m = 1 \rightarrow M$ **do** ▷ Step through each block of butterflies and do twiddle factor m.
      for $i = m \rightarrow N$ **step** $Istep$ **do**
        $j \leftarrow i + M$ ▷ Index i = sum "wing", j = difference "wing."
        $T_{emp} \leftarrow w_m * x_j$ ▷ The start of the butterfly operation; twiddle factor multiplication.
        $x_j \leftarrow x_i - T_{emp}$ ▷ Difference wing of butterfly
        $x_i \leftarrow x_i + T_{emp}$ ▷ Sum wing of butterfly
      end for
    end for
    $M \leftarrow Istep$ ▷ Onto next level!
  end while
end procedure

We test this algorithm by performing the FFT on a square wave signal of magnitude 1 and two full periods, as shown in Figure 2.

The FFT is carried out on a 32-sample test case using 64-bit double precision and the real and imaginary components are plotted in Figure 3. Since the input signal is real, the FFT output will have a a real component that displays even symmetry and the imaginary component will be odd. Since the input signal exhibits nearly odd symmetry, the imaginary component of the transform will dominate. However, the input signal has a tiny bit of even symmetry (the sample $X_0 = 1$, which is by definition even), so there will also be a small real component to the fourier transform. The figure confirms this.

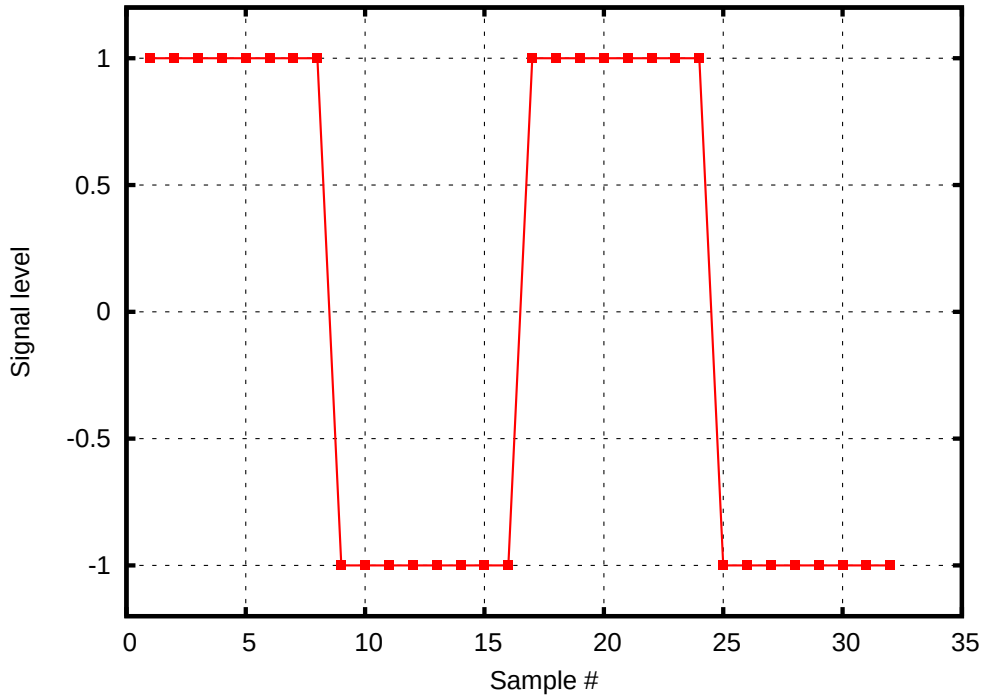

Fig. 2. Input data to 32 point FFT.

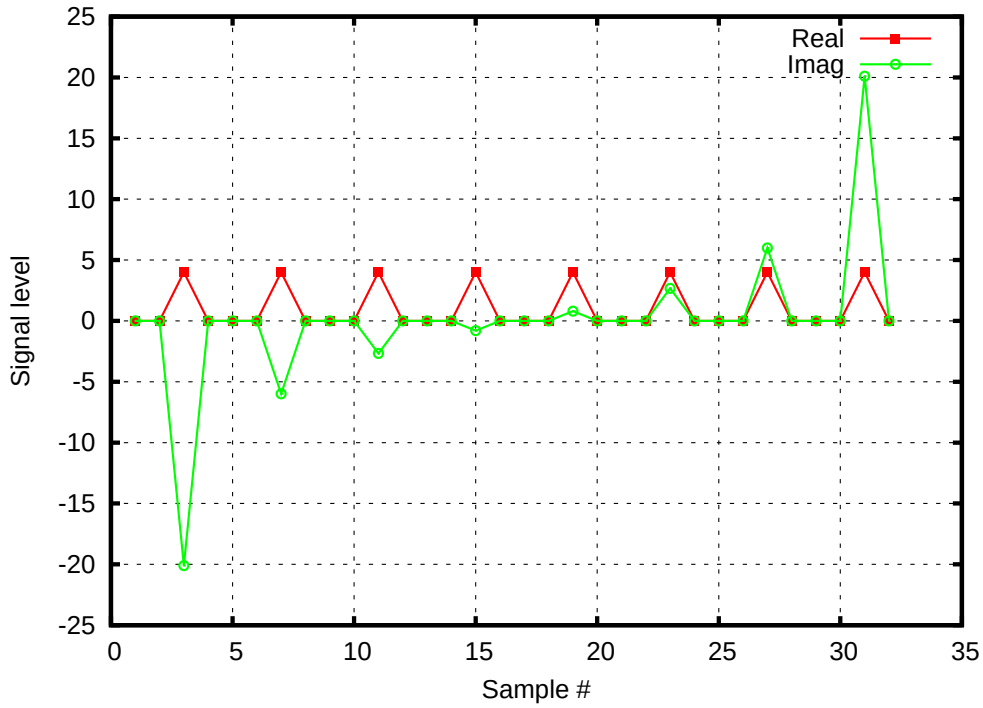

Fig. 3. 32 point FFT output data. Note that the proper symmetries are present and the coefficients are scaled by 32 in this example (using Gnu Octave fft function [10]).

III. The FFT in hardware

When constructing the FFT in hardware, there are a number of significant differences to consider with respect to the software FFT. First and foremost is the fact the a hardware FFT can have many processes going on in parallel, whereas the software FFT (generally) steps through a single instruction at a time. For this reason, hardware FFT processors can have throughputs orders of magnitudes above those of sequential CPUs. This parallel activity means careful thought needs to go into pipelining and timing so data is processed in time for the next stage to receive it. Other differences include the extensive use of integer arithmetic instead of the usual double precision floating-point and being aware of the often limited resources available for mathematical functions in FPGA hardware.

In this case study, we implement a 32 point FFT in hardware using 11 bit signed integer input data. Signed integer arithmatic is used throughout the processor. This is in stark contrast to the use of double precision floating point arithmetic in the software version of the FFT in the previous section. We choose a 32 point FFT in order to show clearly the patterns that one would use to generate longer FFTs without having to cope with long streams of data that would obscure what we wish to show. At the end, we show the results from a larger working implementation in FPGA (a 1024 point FFT with 12 bit width).

A typical hardware FFT processor might be defined as in Figure 4.

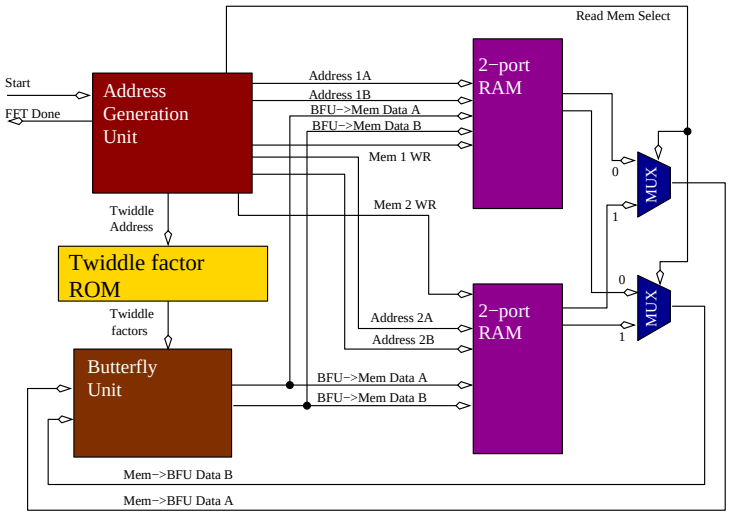

Fig. 4. Top level block diagram of hardware FFT processor. Note the system blocks listed previously: address generator, butterfly unit, memory and twiddle table.

The Address Generation Unit (AGU) controls the generation of addresses for reading and writing the memory contents to and from the Butterfly Processing Unit (BFU). The AGU also generates signals that control writes to memory as well as which memory bank is read. The reader will note that two blocks of two-port RAM are used in this system. All data busses shown represent complex data transfer (double bit widths to accomodate both real and imaginary values). We read from one RAM block, process through the BFU and write to the other RAM block. The main reason for this lies in the fact that we have only two read and write ports on the

FPGA RAM function. The practical need for pipelining the processing operations precludes the possibility of doing simultaneous writes and reads for each butterfly operation. More will be presented on this later. This "ping-pong" memory scheme is a simple way to keep track of the processing level of our data and given the capacity of modern FPGAs, poses few resource problems for FFTs up to $2^{10} - 2^{14}$ in length. Larger FFTs (length 2^{20} or more) can always use large external memories, if needed. We need two memory banks to perform "ping-pong" reads and writes, because of the pipeline delays and the inability to simultaneously read and write to four different addresses in a single memory bank. The FPGA built in functions usually allow a dual port RAM with ports shown in Figure 5.

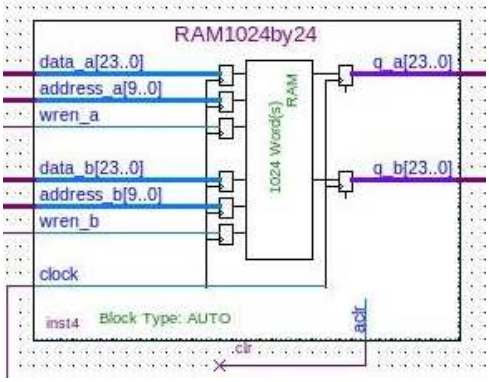

Fig. 5. Synchronous dual port ram as instantiated in FPGA. Only two addresses at a time can be presented to the memory.

The BFU performs a special 2-point FFT on the data pairs specified by the AGU. The atomic operation is schematically shown in Figure 6. A and B are the inputs from the previous level. A' and B' are the outputs after performing the butterfly operation.

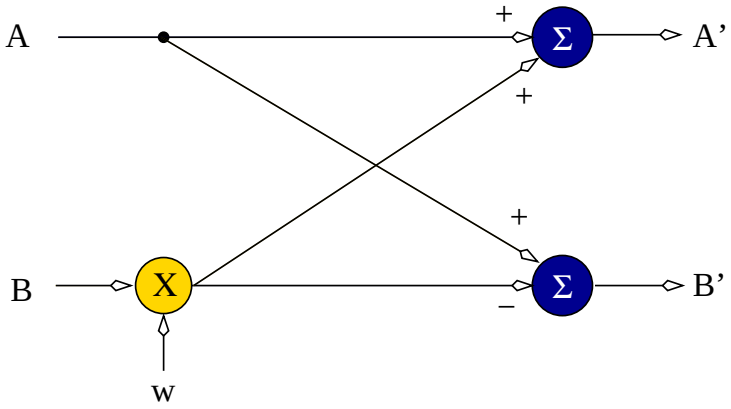

Fig. 6. Description of the BFU operation.

Let us step through the transformation process, describing the action of the various hardware functions. Memory 1 (the top block) is loaded with the data samples to be transformed (in bit-reversed address order) and the $Start$ signal is triggered. The $FFT\ Done$ signal goes low and the AGU starts cycling through the Memory

1 addresses and the twiddle factor addresses as the BFU processing pipeline begins to fill. After a number of clock cycles, data begins to appear at the output of the BFU. The AGU begins to generate write cycles to Memory 2 (the bottom block) and the processed data is written to Memory 2. When the AGU reaches the end of the data buffer, the read address counter stops while the write address counter continues until the BFU pipeline has completely flushed out. Once the output data is completely written, the "level counter" increments and the read address counter and twiddle factor address counter begins to increment in an appropriately permuted order that depends on the level counter value. With this, the whole process repeats until the level counter indicates that we have completed the full transform. When this happens, the $FFT\ Done$ signal is asserted, the BFU pipeline is flushed and the whole FFT processor goes into a wait state. The results of the transform can now be read out and new data samples can be written into the memory. The $Start$ line is triggered and the next batch of data is transformed.

A. The butterfly processor

The BFU is a straightforward implementation of the mathematical operation seen in Figure 6. Its block diagram is seen in Figure 7

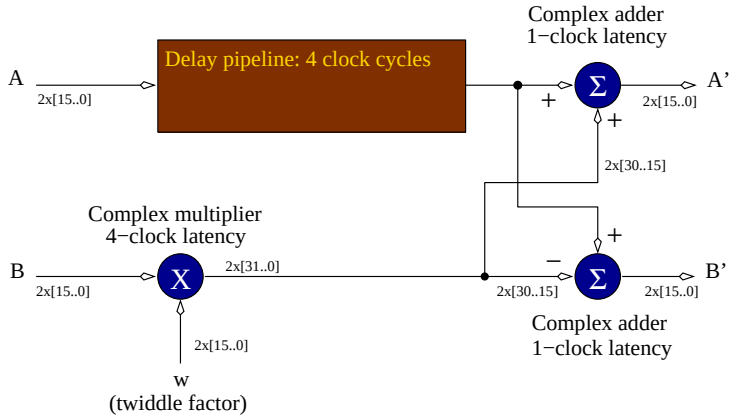

Fig. 7. Block diagram of the hardware implementation of the BFU. Note the timing latencies that are present in the practical implementation that the conceptual version does not include. Output data appears 5 clock cycles after data is presented on the input busses.

In the practical implementation of the BFU, we need to include the effects of finite time latencies needed to perform the multiplications and additions. The A arm contains no multiplication and needs a four clock delay to align the data at the adders to properly generate the A' and B' sums. Notice that although we use 11 bit signed real and imaginary data on the input, the BFU input and output data consist of two data busses (real and imaginary) that are 16 bits wide. **The BFU needs 5 extra bits to accommodate "bit growth" that occurs as the FFT processor cycles through the butterfly levels!** This is critical to preserving precision, since we are doing all computations using signed integer arithmetic. At the end of the FFT, we can always prune the lowest order bits to reduce the bit width on the output. (An alternative method would be to perform bit truncation and rounding after each FFT level, but the loss of precision is slightly worse than accomodating growth with extra bits and a bit of extra latency can be expected as well.)

Note also that the multiplication of 16 bit numbers produces 32 bit products. Signed integer multiplication also has the interesting property of producing redundant sign bits in the product ([7] as long as we are not multiplying two maximum magnitude negative numbers). Hence, we route bits [30..15] from the multiplier to the adders, in effect performing a left-shift on the data bits, otherwise the magnitude will not be correct.

Making the full FFT system work requires properly accounting for the inherent pipeline latency of the BFU. Pipelining is an indispensible tool that permits high speed digital processing at the cost of adding latency to the output. By breaking up complicated tasks (like complex multiplication) into smaller chunks, we avoid the problem of uneven delays in combinatorial logic potentially spoiling the data. All data is guaranteed to be present on the output after a well defined number of clock cycles regardless of the input conditions.

B. Review of integer number system

All high level microprocessors found in personal computers have built in floating point processing units that greatly facilitate arithmetic operations on 32 and 64 bit wide floating point numbers. This simplifies the implementation of fast numerical methods, rescaling is automatice and the data is in a “human friendly” form (scientific notation). When implementing fast digital signal processing algorithms in hardware, floating point numbers have several disadvantages.

- Large word width occupies many memory cells.
- Arithmetic operations on floating point numbers are much more complex than on fixed point or integer numbers. Many logic cells are required.
- Speed and or latency is degraded because of extra complexity.
- Since digitized signals have fixed word width, floating point offers no processing advantage other than being easy for humans to recognize.

In this paper, we use 16-bit fixed point signed fractional arithmetic. Numbers are represented as

$$x = s.d_{14}d_{13}d_{12}d_{11}d_{10}d_9d_8d_7d_6d_5d_4d_3d_2d_1d_0 \tag{17}$$

where s is the sign bit and d represents the mantissa bits for each power of 2 using the usual 2s complement signed number system.

Unlike the usual floating point system, there is no rescaling that takes place after performing an operation. Adding two 16 bit numbers produces a 17 bit result, whereas multiplication of two 16 bit numbers yields a 32 bit result. The designer must take care that overflows or underflows do not occur during processing. This means that word widths must be appropriately chosen and scaling (rounding, word truncation) must be appropriately used to keep numbers within the required limits so that numerical significance is maximized.

C. The address generation unit

This is the most challenging part of the FFT processor. We need to generate addresses for the reading and writing of data RAM, retrieve twiddle factors and generate write signals for the data RAM. Furthermore, we need to keep track of which butterfly we are executing as well as which FFT level we are working on. Let us start with the sweep through the butterfly operations.

A classic short early paper on hardware FFT implementation [8] shows us an elegant strategy for generating the addresses of the pairs of data points for the butterfly operation. A straightforward stepping through the butterfly pairs is hinted at by the software FFT. They go as such:

Iteration level i $\downarrow$	Butterfly address pairs $j \longrightarrow$
Level 0:	$\{0,1\}\ \{2,3\}\ \{4,5\}\ \{6,7\}\ \{8,9\}\ \cdots$
Level 1:	$\{0,2\}\ \{1,3\}\ \{4,6\}\ \{5,7\}\ \{8,10\}\ \cdots$
Level 2:	$\{0,4\}\ \{1,5\}\ \{2,6\}\ \{3,7\}\ \{8,12\}\ \cdots$
Level 3:	$\{0,8\}\ \{1,9\}\ \{2,10\}\ \{3,11\}\ \{4,12\}\ \cdots$
$\vdots$	

for implementing a radix-2 DIT FFT.

It turns out that by reordering the butterfly pairs, as such:

Iteration level i $\downarrow$	Butterfly address pairs $j \longrightarrow$
Level 0:	$\{0,1\}\ \{2,3\}\ \{4,5\}\ \{6,7\}\ \{8,9\}\ \cdots$
Level 1:	$\{0,2\}\ \{4,6\}\ \{8,10\}\ \{12,14\}\ \{16,18\}\ \cdots$
Level 2:	$\{0,4\}\ \{8,12\}\ \{16,20\}\ \{24,28\}\ \{1,3\}\ \cdots$
Level 3:	$\{0,8\}\ \{1,9\}\ \{2,10\}\ \{3,11\}\ \{4,12\}\ \cdots$
$\vdots$	

produces a simple function between the pair indices i, j and the pair addresses $\{m, n\}$ and does not change the final result. It is a simple process to verify that each address pair is given in terms of the indices by

$$m = \textbf{Rotate}_5(2j, i) \tag{18}$$

$$n = \textbf{Rotate}_5(2j+1, i) \tag{19}$$

where $\textbf{Rotate}_N(x, y)$ indicates a circular left shift of N bit word x by y bits.

The twiddle factor addresses are found by masking out the $N-1-i$ least significant bits of j. For the length 32 FFT, the twiddle factor table is given Table II.

We can get an idea of how the FFT processor works by implementing the AGU and BFU as a C program snippet using integer arithmetic.

On entering this program, the arrays $Data_r$ and $Data_i$ contain the real and imaginary parts of the samples in **bit-reversed order**. The arrays Tw_r and Tw_i contain the lookup table of twiddle factors. Since this code is run on a personal computer, the integer bit size is 32 bits (instead of the 16 bits we use in our example). This causes no problems because we just limit our input data to the 11 bit size and use the required sign extension on the bits we choose to ignore.

The outer for loop steps through the levels i and the second for loop steps through each butterfly index pair within the level. Lines 6-9 generate the addresses by first using a left-shift to perform an integer multiply-by-2

TABLE II

TABLE OF TWIDDLE FACTORS GIVEN AS FLOATING POINT DECIMAL AND 16-BIT HEXADECIMAL SIGNED INTEGERS.

Address k	$\cos(2\pi k/32)$ float	$\cos(2\pi k/32)$ 16-bit integer	$\sin(2\pi k/32)$ float	$\sin(2\pi k/32)$ 16-bit integer
0	1.0000e+00	0x7fff	0	0
1	9.8079e-01	0x7d89	1.9509e-01	0x1859
2	9.2388e-01	0x7641	3.8268e-01	0x30fb
3	8.3147e-01	0x6a6d	5.5557e-01	0x471c
4	7.0711e-01	0x5a82	7.0711e-01	0x5a82
5	5.5557e-01	0x471c	8.3147e-01	0x6a6d
6	3.8268e-01	0x30fb	9.2388e-01	0x7641
7	1.9509e-01	0x18f9	9.8079e-01	0x7d89
8	0	0x0	1.0e+00	0x7fff
9	-1.9509e-01	0xe707	9.8079e-01	0x7d89
10	-3.8268e-01	0xcf05	9.2388e-01	0x7641
11	-5.5557e-01	0xb8e4	8.3147e-01	0x6a6d
12	-7.0711e-01	0xa57e	7.0711e-01	0x5a82
13	-8.3147e-01	0x9593	5.5557e-01	0x471c
14	-9.2388e-01	0x89bf	3.8268e-01	0x30fb
15	-9.8079e-01	0x8277	1.9509e-01	0x1859

to produce the first index. We then add '1' to the first index result to get the second. To generate the actual data addresses, we need to perform a circular shift. There is no C instruction for doing circular shifts explicitly, so we need to invent a way to do this. Lines 8 and 9 use a combination of left and right logical shifts to simulate the rotate operation over the 5 bit address. We then apply a masking operation to zero out bits [31:5] in the integer word (otherwise we will generate segmentation faults). The variables ja and jb now contain the addresses of the A and B butterfly values.

The twiddle factor address is computed using a right shift and masking operation on the j index as outlined by [8]. This data is then used to perform the butterfly operation on the integer data set. Let us take a look at the sequencing of the data addresses and the twiddle factor addresses generated with this code. We have verified the method and Table IV gives us the address sequences that we expect from our hardware generator.

The full address generator unit (AGU) is shown in Figure 8.

Triggering the $Start\ FFT$ line sets the synchronous SR latch (third flip-flop from the bottom left) and asserts the $ClearHold$ signal to reset all storage elements to a predictable "0" state for two clock cycles. After two clock cycles $ClearHold$ goes low and the address counter (top left blue block) begins to count at the system clock rate. The output of the address counter is hard-wired to give the shift-left-by-1 so we have the multiply-by-2 for the even indices and multiply-by-2 plus 1 for the odd ones. These values are fed through the rotate-left blocks (where the amount of the rotate is determined by the level counter, now at zero). The red clock delay blocks are needed to synchronize the data passing through the arms where no pipelined arithmetic operation is needed; i.e., so all addresses are lined up properly to send to the data RAM blocks.

The twiddle factor look-up table address is computed directly from the memory counter output. The value

TABLE III

LISTING OF AGU AND BFU IN C WITH INTEGER ARITHMETIC

```
1    for(i = 0; i < 5; i++)  // Level of FFT
2    {
3      for(j = 0; j < 16; j++) // Butterfly index
4      {
5  /* Generate addresses for data and twiddles. */
6         ja = j << 1;  // Multiply by 2 using left shift.
7         jb = ja + 1;
8         ja = ((ja << i) | (ja >> (5 - i))) & 0x1f; // Address A; 5 bit circular left shift
9         jb = ((jb << i) | (jb >> (5 - i))) & 0x1f ; // Address B; implemented using C statements
10
11        TwAddr = ((0xffffff0 >> i) & 0xf) & j; // Twiddle addresses
12
13 /* Do the butterfly operation on the data. */
14        temp_r = ((Data_r[jb] * Tw_r[TwAddr]) / 32768)  // Divide by 32768 (2^15)
15           - ((Data_i[jb] * Tw_i[TwAddr]) /32768);       // does a 16-bit right arithmetic shift
16        temp_i = ((Data_r[jb] * Tw_i[TwAddr]) / 32768)  // on the product
17           + ((Data_i[jb] * Tw_r[TwAddr]) / 32768);      // data.
18        Data_r[jb] = Data_r[ja] - temp_r ;
19        Data_i[jb] = Data_i[ja] - temp_i ;
20        Data_r[ja] += temp_r;
21         Data_i[ja] += temp_i;
22       }
23    }
```

TABLE IV

THE SEQUENCE OF ADDRESSES GENERATED USING THE COUNT AND ROTATE TECHNIQUE.

Index j	Level 0		Level 1		Level 2		Level 3		Level 4	
i	ja	jb	ja	jb	ja	jb	ja	jb	ja	jb
0	0	1	0	2	0	4	0	8	0	16
1	2	3	4	6	8	12	16	24	1	17
2	4	5	8	10	16	20	1	9	2	18
3	6	7	12	14	24	28	17	25	3	19
4	8	9	16	18	1	5	2	10	4	20
5	10	11	20	22	9	13	18	26	5	21
6	12	13	24	26	17	21	3	11	6	22
7	14	15	28	30	25	29	19	27	7	23
8	16	17	1	3	2	6	4	12	8	24
9	18	19	5	7	10	14	20	28	9	25
10	20	21	9	11	18	22	5	13	10	26
11	22	23	13	15	26	30	21	29	11	27
12	24	25	17	19	3	7	6	14	12	28
13	26	27	21	23	11	15	22	30	13	29
14	28	29	25	27	19	23	7	15	14	30
15	30	31	29	31	27	31	23	31	15	31

is the logical AND of the 4-bit counter output and the twiddle mask generator. (The twiddle mask generator is a right-shift register that fills up with "1"s as the level counter is incremented.)

When the address counter overflows at 15, it triggers a delayed increment of the level counter and sets the hold-counter trigger flip-flop. This holds the address counter in a cleared state until the hold counter times out. The purpose of this is to allow the BFU and RAM write pipelines to flush out before we go to the next level in the FFT. In this way we prevent data reads on the next level before all data from the previous level is properly written to data RAM.

When both the address counter and the level counter overflow, we know we have finished the FFT and we can stop. The $FFT\ Done$ line is asserted and the processing stops. At this point we can read in new data and read out the FFT data. We then retrigger the $Start\ FFT$ line and the whole process repeats itself.

Simulations (to be presented in detail later) verify the address pattern is correct.

D. The data memory structure

In order to do anything useful, we need to be able to store the data we wish to transform and hold the intermediate values as we step through the FFT levels. To do this we design a random access memory block. In reality, there are four blocks, as seen in Figure 9.

Perhaps it is easiest to step through the various input and output variables as a list.

- Inputs:

  $LoadDataWrite$

  This signal, when pulled high, enables writing of new data to the memory as well as reading out the results of the previously executed transform.

  $Bank0WriteEN$

  Enable writes to Bank 0 memory block. Note that reads are always possible from any memory block.

  $Bank1WriteEN$

  Enable writes to Bank 1 memory block.

  $Data\_real\_in[15..0]$

  Real part of input data. Data is 16 bits wide.

  $Data\_imag\_in[15..0]$

  Real part of input data.

  $RWAddrEN$

  This signal to the address MUXs switches between $ReadAddr[4..0]$ and $WriteAddr[15..0]$ inputs.

  $BankReadSelect$

  Select line that controls source of data reads to be fed to the BFU. Memory blocks toggle back-and-forth between levels and this line is needed to allow the toggling to take place.

  $LoadDataAddr[4..0]$

  When memory is filled with new data, the data is written to the addresses given on these 5 lines. Data at the addresses presented on these lines is also output on H_real and H_imag.

$ReadAddr[4..0]$

Addresses for data reads of G and H.

$WriteAddr[4..0]$

Addresses for data writes. Must be appropriately delayed for proper data alignment.

$Xr[15..0]$

Real part of data to be written to A data block (from BFU-¿Mem).

$Xi[15..0]$

Imaginary part of data to be written to A block.

$Yr[15..0]$

Real part of data to be written to B block.

$Yi[15..0]$

Imaginary part of data to be written to B block.

- Outputs:

G_real

A block real output to BFU.

G_imag

A block imaginary output.

H_real

B block real output.

H_imag

B block imaginary output.

Note that the total latency of the memory system is 4 clock cycles from the time that addresses/data are presented on the inputs until the data appears on the outputs. This structure is completely scalable by changing the address and data widths (and memory sizes) as required.

The top level system block diagram is visible in Figure 10.

Starting from the left of the figure, the twiddle factor ROM contains the look-up table of real and imaginary values of the required “roots of unity” that are passed to the BFU. The AGU, as described earlier, generates all twiddle factor and data memory addresses in the proper sequence and indicates when the outputs of the BFU are written to memory. Some delay blocks are needed to ensure that memory reads and writes occur at the correct time. We also see a block indicating the bit reversal operation on the data address input (when new data is written to data RAM, it is always stored in bit-reversed order). The two flip-flops at the top right control the memory bank select. The left flip flop is a delay/edge detector, whereas the right flip-flop is toggles the memory bank on each memory write assertion. We will have a closer look at this action in the next section.

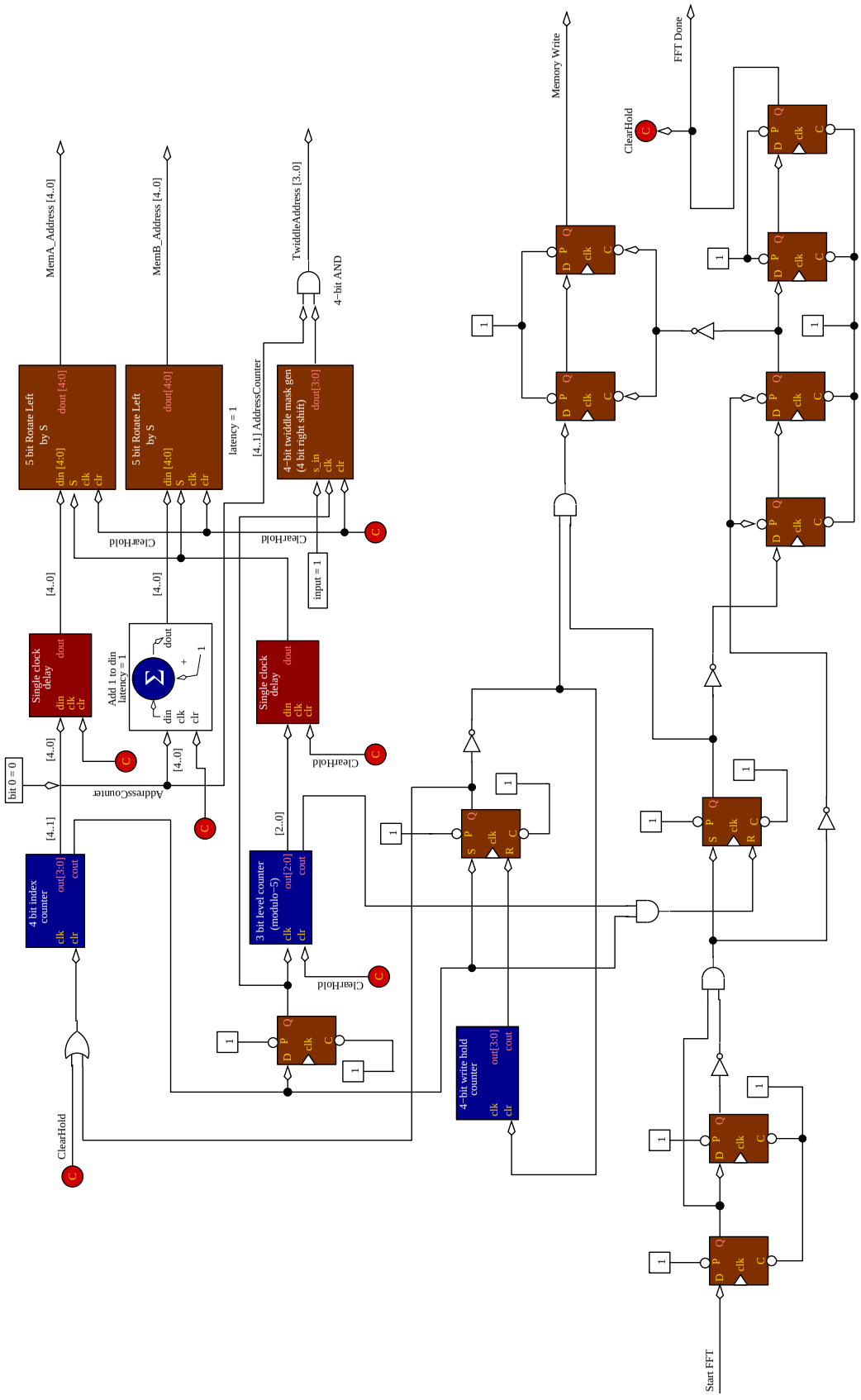

Fig. 8. Block diagram of the hardware implementation of the AGU. Note that all flip-flops, counters, shift registers are synchronously clocked by a global system clock. Connections are left out to improve clarity in the diagram.

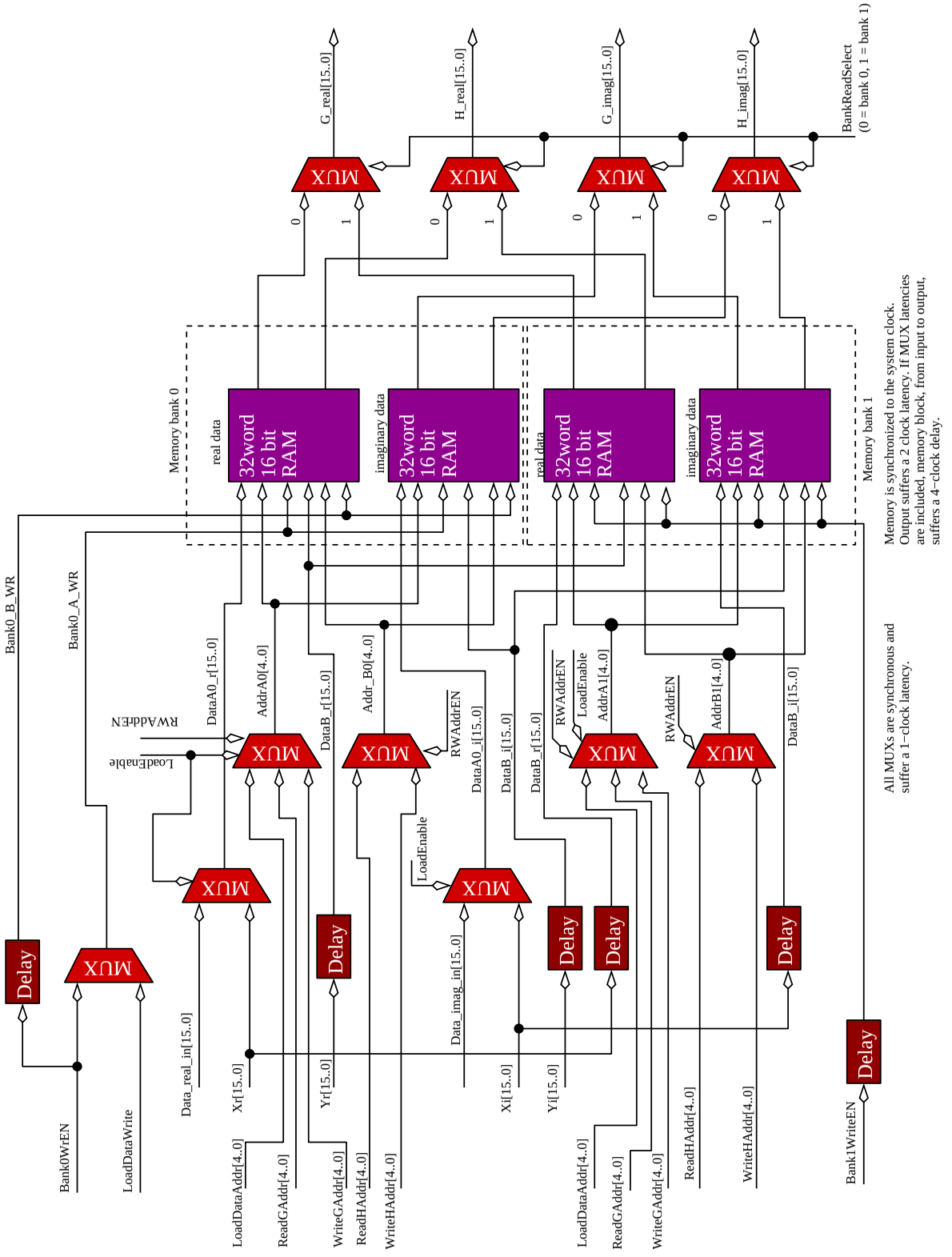

Fig. 9. Memory block diagram. Note the division into two banks of “real” and “imaginary” storage with multiplexers that control routing of signals.

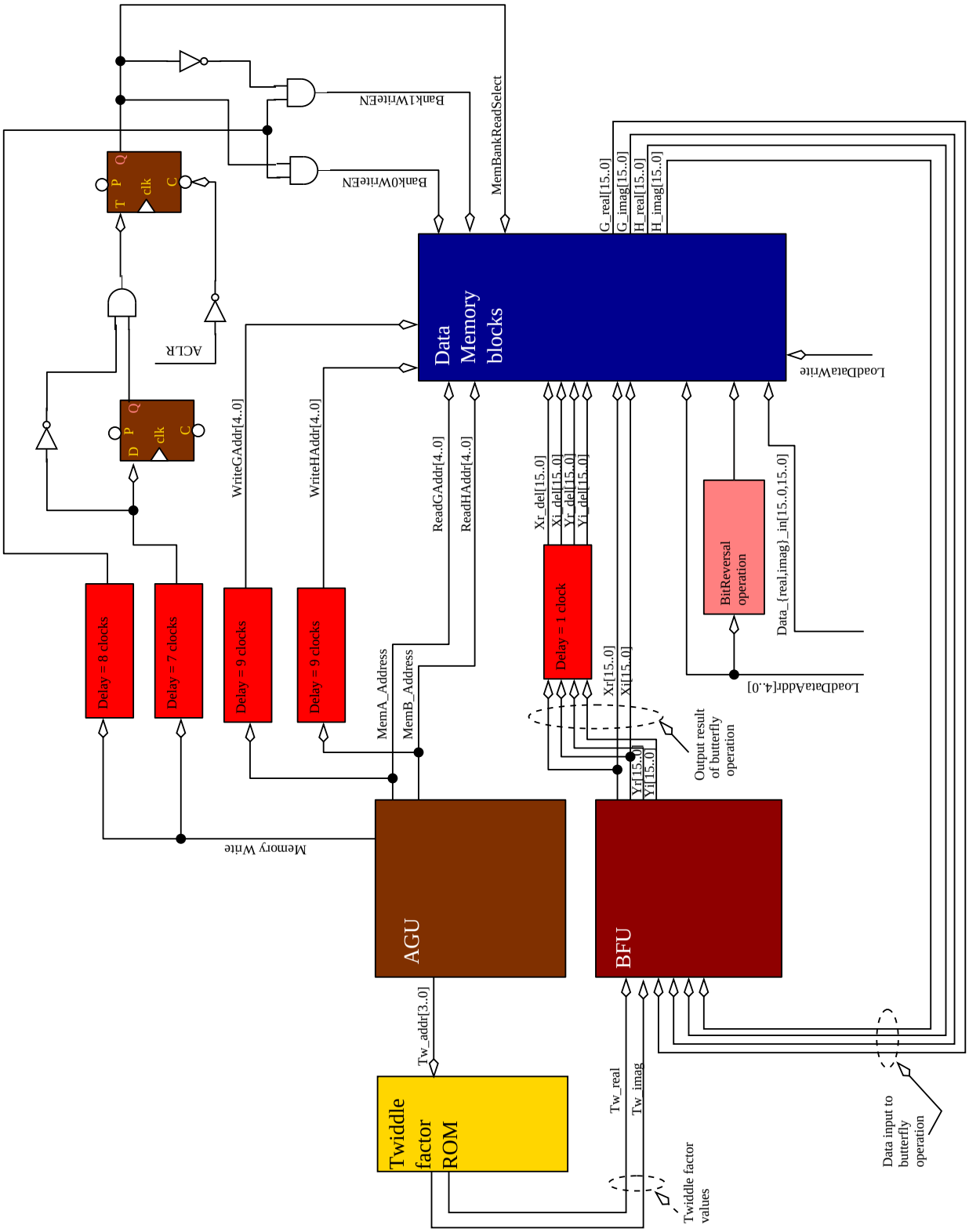

Fig. 10. Full system block diagram showing the added pipeline delays needed to properly synchronize data flows into the BFU and data memory. System clock is implied.

IV. EXAMPLE FFT CALCULATION

In order to demonstrate the correctness of the hardware implementation, we use the input data set shown in Figure 2. The floating point data set is converted to a fixed point integer representation where 1.0 $\rightarrow$ 1023 (or 0x3ff in 16 bit representation) and -1 $\rightarrow$ -1023 (0xfc01 in 16 bit two's complement hexidecimal notation). Thus we get the waveforms in Figure 11

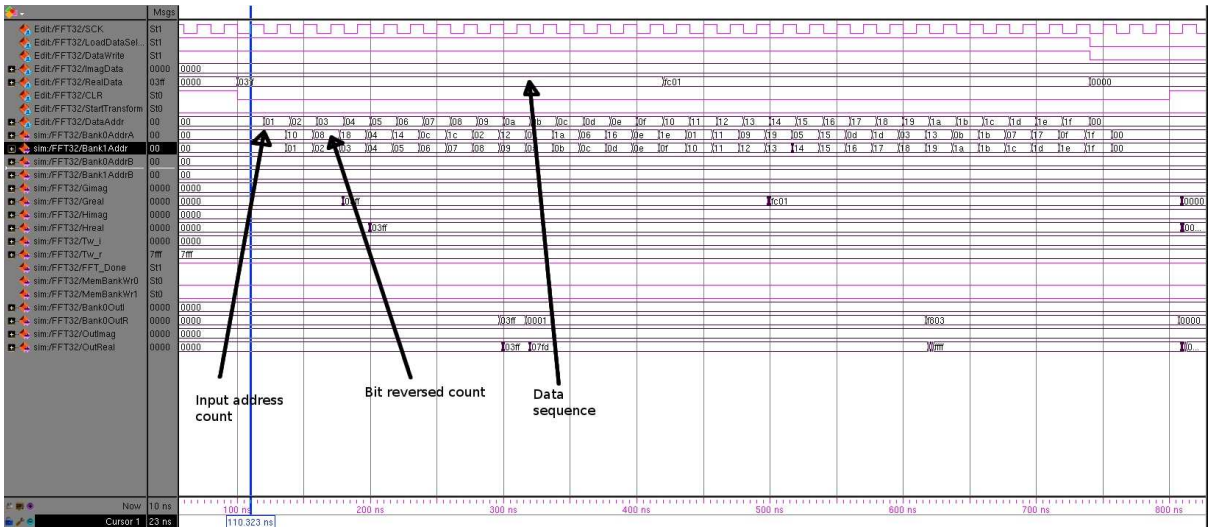

Fig. 11. Waveforms showing sequence of input data.

The signal labeled $DataAddr$ is aligned with the data signals $RealData$ and $ImagData$. The signals $Bank0Addr$ and $Bank1Addr$ are the bitswapped and non-bitswapped addresses that appear at the input to the memory blocks. Notice the 1-clock delay with respect to the input addresses. This delay is the latency of the input address multiplexer.

The imaginary component of the input data is set to zero; only the real component is non zero, as in the initial example.

The next figure shows how the FFT is initiated once the all the input data is entered. We first trigger the CLR input to ensure that the AGU is in a cleared state (although it generally already will be waiting because any spurious running state will time out quickly on power-up or by an initial triggering of a general system clear just after start up). We then trigger the $StartTransform$ line, as seen in Figure 12.

The memory read process starts 9 clock cycles after the start trigger is issued. This is the amount of time needed for the address pipeline to fill.

Once the read process starts, we need to wait until the data pipeline fills before we start writing to memory. Hence the delay in the memory addresses and write enable signal in Figure 13

Bank 1 write is active indicating that the reads are taking place from Bank 0, being processed by the BFU and are being written to Bank 1. This continues until all 16 pairs are written to Bank 1. The Bank 1 write enable $MemBankWr1$ goes low and writes are disabled until the next write cycle is initiated for the next level.

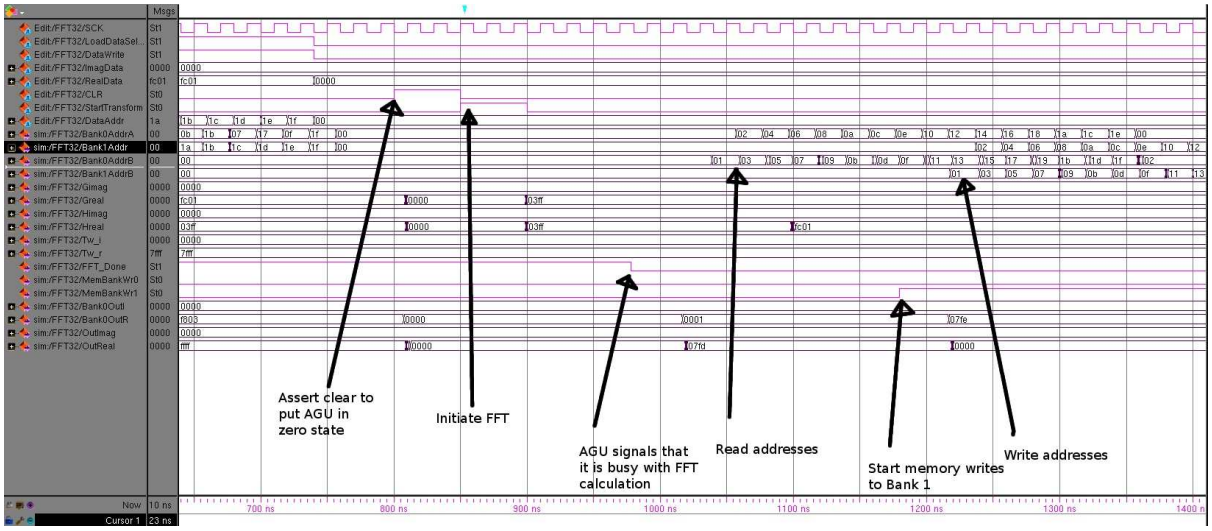

Fig. 12. Illustration of the transform initiation sequence.

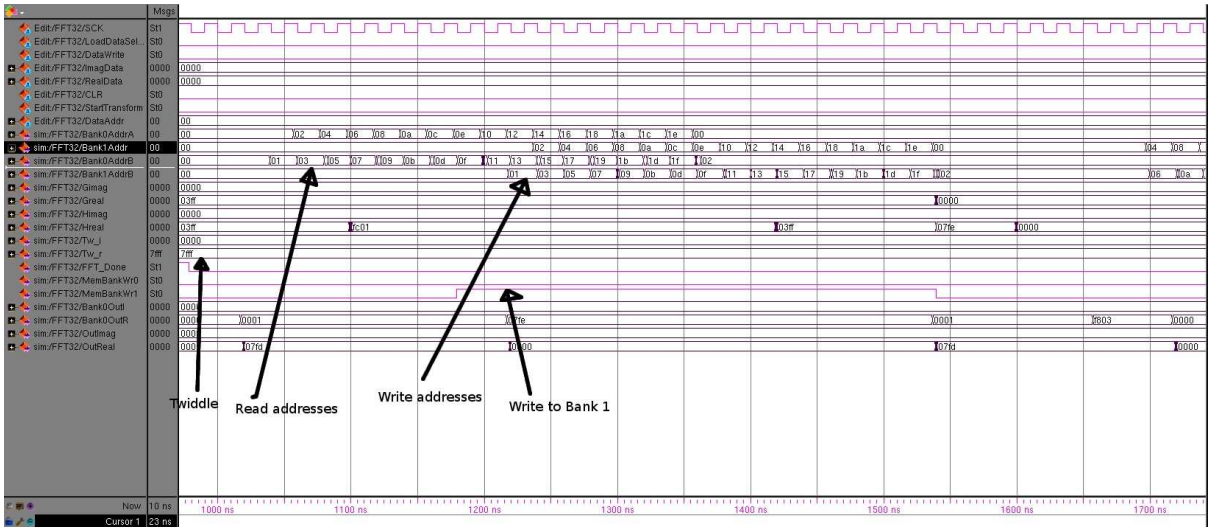

Fig. 13. Illustration of the first level of computations for the FFT.

The second level of FFT computations starts with the read addresses following the correct modified progression $\{0, 2\}$, $\{4, 6\}$, $\{8, a\}$, $\cdots$, seen in Figure 14.

Notice the positioning of the data (output data from the BFU) with respect to the write addresses and the write-enable signal. We see that the timing is correct. Also, we see the "ping-pong" memory addressing in action. We are now reading data from Bank 1 and writing to Bank 0.

This pattern repeats until we reach the last level (level 5). When the read address counter times out, $FFTDone$ goes high indicated that the computation is complete and the BFU pipeline is permitted to flush out (writing the final results to memory. Figure 15 shows the final results as they are written to memory, listed in Table V in fixed point and compared to the floating point result. The comparison with the data generated using floating

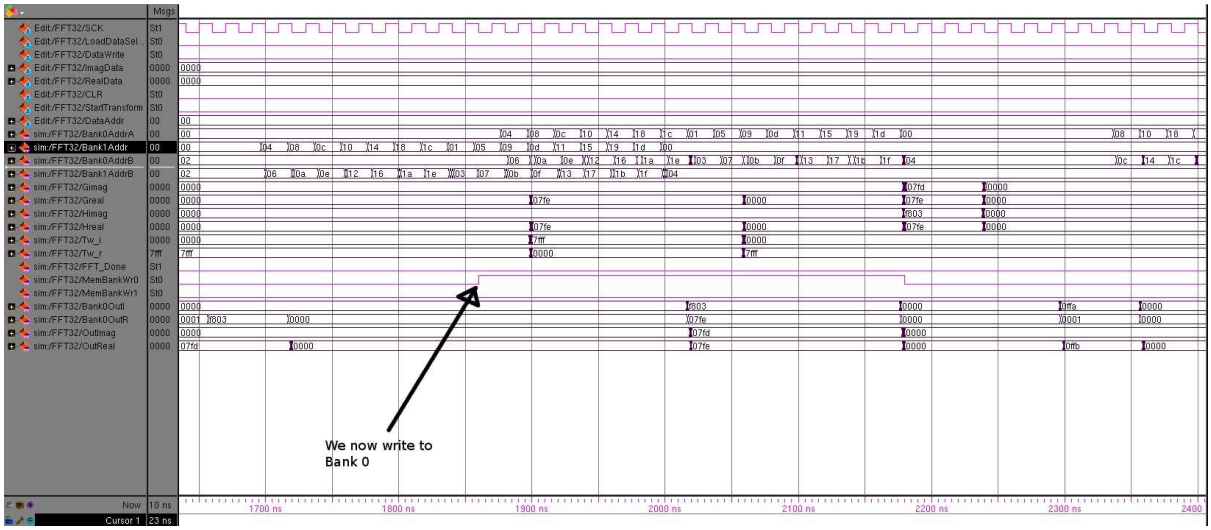

Fig. 14. Illustration of the second level of computations for the FFT.

point arithmetic in Octave shows a good preservation of precision in the fixed point results.

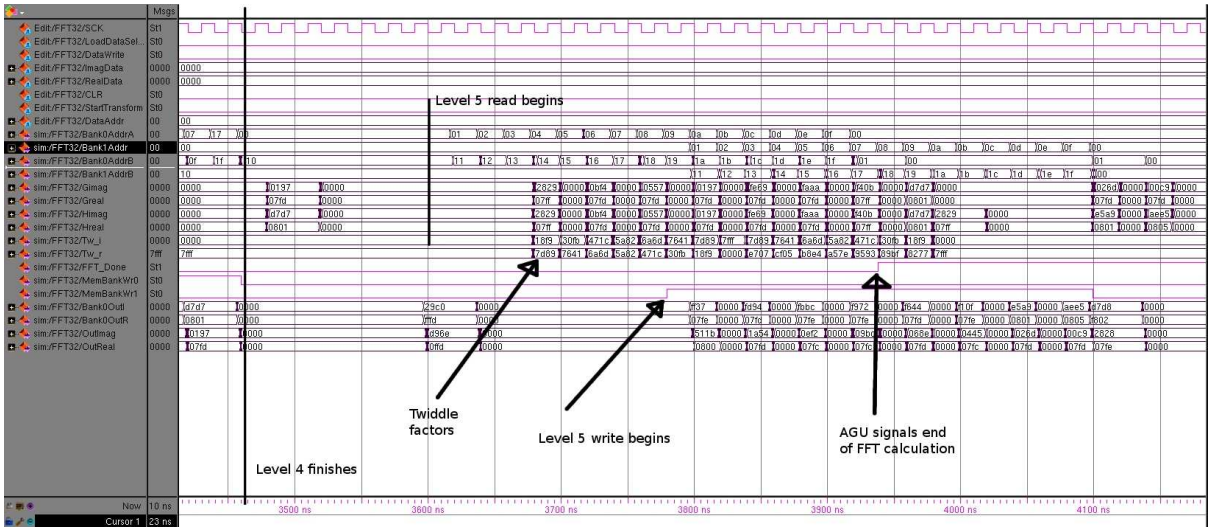

Fig. 15. Final data as it is written to Bank 1 is present in the bottom four waveforms. This data is reproduced ans compared to Octave FFT result in V.

TABLE V

RESULTS OF HARDWARE FFT. DATA IS PRESENTED IN FIXED-POINT REPRESENTATION AND THE EQUIVALENT FLOATING POINT REPRESENTATION SCALED TO UNITY.

Index	HW FFT (fixed-point)		HW FFT (equiv. float)		FFT (Octave-float)	
	real	imag	real	imag	real	imag
0	0x0	0x0	0	0	0	0
1	0x0800	0x511b	6.25×10^{-2}	6.34×10^{-1}	6.25×10^{-2}	6.35×10^{-1}
2	0x0	0x0	0	0	0	0
3	0x07fd	0x1a54	6.24×10^{-2}	2.05×10^{-1}	6.25×10^{-2}	2.06×10^{-1}
4	0x0	0x0	0	0	0	0
5	0x07fc	0x0ef2	6.24×10^{-2}	1.17×10^{-1}	6.25×10^{-2}	1.17×10^{-1}
6	0x0	0x0	0	0	0	0
7	0x07fc	0x09bc	6.24×10^{-2}	7.60×10^{-2}	6.25×10^{-2}	7.61×10^{-2}
8	0x0	0x0	0	0	0	0
9	0x07fd	0x068e	6.24×10^{-2}	5.12×10^{-2}	6.25×10^{-2}	5.13×10^{-2}
10	0x0	0x0	0	0	0	0
11	0x07fc	0x0445	6.24×10^{-2}	3.34×10^{-2}	6.25×10^{-2}	3.34×10^{-2}
12	0x0	0x0	0	0	0	0
13	0x07fd	0x026d	6.24×10^{-2}	1.90×10^{-2}	6.25×10^{-2}	1.90×10^{-2}
14	0x0	0x0	0	0	0	0
15	0x07fd	0x00c9	6.24×10^{-2}	6.13×10^{-3}	6.25×10^{-2}	6.16×10^{-3}
16	0x0	0x0	0	0	0	0
17	0x07fe	0xff37	6.24×10^{-2}	-6.13×10^{-3}	6.25×10^{-2}	-6.16×10^{-3}
18	0x0	0x0	0	0	0	0
19	0x07fd	0xfd94	6.24×10^{-2}	-1.90×10^{-2}	6.25×10^{-2}	-1.90×10^{-2}
20	0x0	0x0	0	0	0	0
21	0x07fe	0xfbbc	6.24×10^{-2}	-3.33×10^{-2}	6.25×10^{-2}	-3.34×10^{-2}
22	0x0	0x0	0	0	0	0
23	0x07fe	0xf972	6.24×10^{-2}	-5.12×10^{-2}	6.25×10^{-2}	-5.13×10^{-2}
24	0x0	0x0	0	0	0	0
25	0x07fd	0xf644	6.24×10^{-2}	-7.60×10^{-2}	6.25×10^{-2}	-7.61×10^{-2}
26	0x0	0x0	0	0	0	0
27	0x07fe	0xf10f	6.24×10^{-2}	-1.17×10^{-1}	6.25×10^{-2}	-1.17×10^{-1}
28	0x0	0x0	0	0	0	0
29	0x0801	0xe5a9	6.25×10^{-2}	-2.06×10^{-1}	6.25×10^{-2}	-2.06×10^{-1}
30	0x0	0x0	0	0	0	0
31	0x0805	0xaee5	6.26×10^{-2}	-6.34×10^{-1}	6.25×10^{-2}	-6.35×10^{-1}

V. SUMMARY AND CONCLUSION

As is the case with many technical and scientific problems, it is often a good strategy to start off with first principles and build up the edifice on a good foundation of understanding. Consistent with this philosophy, we review the basics of the FFT starting from the DFT as a linear transformation of a vector containing a set of sample data. From there, we show how exploiting the symmetries of the DFT linear transform produces the FFT. It is then a short step to assemble a floating-point algorithm (like you might run on a PC). We then

develop a fixed point (integer) algorithm to demonstrate the address generation method and to illustrate the awareness one needs for scaling of the FFT result.

The construction of the FFT in hardware starts off from the same theoretical base as the software transform, but requires a significantly different approach to its implementation. In most cases, the software transform is designed to run on a microprocessor that typically steps through a set of instructions one by one. The processing is organized in a serial fashion and is readily described using standard programming languages. The hardware FFT, in contrast, is usually designed to perform its component tasks with at least some degree of parallelism. This leads to a significantly different method of algorithm construction with respect to a microprocessor implementation.

In this document, we describe the construction of the Cooley-Tukey Decimation-In-Time algorithm for implementation in an FPGA (See Figure 16). A high degree of parallelism is built into the transform such that arithmetic processing, memory fetches and writes occur simultaneously, thereby speeding up throughput and reducing processing latency over less parallelized versions. We do this by breaking all the operations up into small "chunks" that each perform one small task in a clock cycle and pass the result onto the next stage and retrieving the result of the previous stage. This "assembly line" or "pipelining" philosophy is indispensible to processing data at high speed. Throughput can be orders of magnitude above that of serial instruction microprocessors.

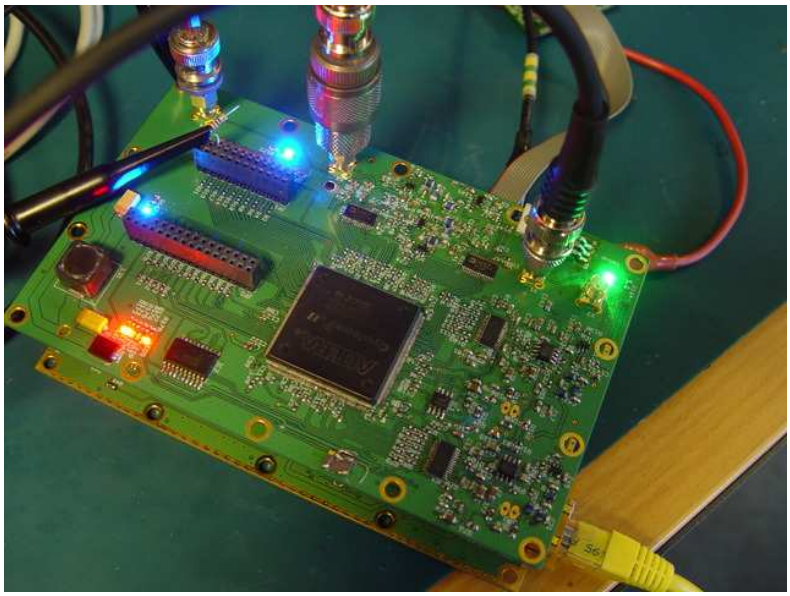

Fig. 16. Example FPGA platform for implementing FFT. Besides the large FPGA chip in the middle, there are two high speed ADCs and a pair of high speed DACs for easy data conversion.

Despite all the simulations that we presented in the preceding section, there is no substitute for compiling the design and loading it onto a real FPGA for testing. This we did for an expanded FFT (1024 points, 12 bit data). Using a 1221 Hz 3.0V peak-to-peak square wave generated by a signal generator, we performed a 50ksample/second digitization. 1221Hz is approximately $25 \times 50000/1024$, hence will generate distinct peaks at frequency points (bins) 25, 75, 125, 175, etc., i.e. the odd harmonics. This is beautifully illustrated in the

results generated by our FPGA FFT in Figure 17.

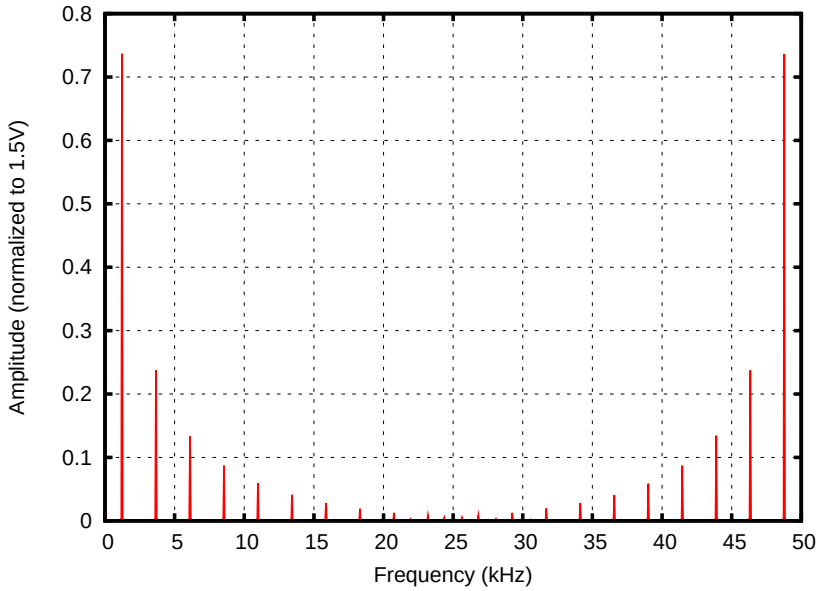

Fig. 17. 1221 Hz square wave spectrum generated by FPGA FFT of digitized signal. The main peak is at 1221Hz (the 20th bin) and the harmonics roll off as 1/n, at the odd harmonic positions 3663, 6105, 8547,$\cdots$ Hz.

The specifications of the system used to generate this plot are enumerated below

- FPGA type: Cyclone II EP2C20
- Sample rate: 50ksps
- Number of points: 1024 (10 levels)
- Signal amplitude: 1.5V (3V p-p full scale)
- Signal frequency: 1221Hz (= 20 times 5000 / 1024, so signal appears periodic at boundaries)
- Analog to digital word: 12 bit, signed two's complement
- FFT memory width 24 bit (22-bits needed to fully accomodate bit growth)
- System clock rate: 50 MHz
- Conversion speed: approx 100us for 1024 point complex transform and 50MHz clock rate.

A hardware FFT opens up a whole range of possibilities for real-time signal processing. Aside from spectral estimation of real-time signals, complicated filtering tasks can be made as simple as "drawing the frequency response." Block convolution and correlation is greatly simplified (and speeded up) with respect to multiply-and-shift type algorithms. Tasks like carrier acquisition and OFDM modulation/demodulation can be carried out with ease. I leave it up to the reader to dream up other applications for this design.

REFERENCES

[1] S. Winograd, "On computing the discrete Fourier transform," *Math. Comp.*, 32, pp. 175-199, 1978.

[2] S. Winograd, "On the multiplicative complexity of the discrete Fourier transform," *Adv. Math.*, 32, pp. 83-117, 1979.

[3] N. Brenner and C. M. Rader, "A new principle for fast Fourier transformation," *IEEE Acoustics Speech and Sig. Proc.*, no. 24, vol. 3, pp. 264-266.

[4] I. Good, “The interaction algorithm and practical Fourier analysis,” *J. R. Statist. Soc. B*, no. 20. vol. 2, pp. 361-372.

[5] L. R. Rabiner, R. W. Schaefer and C. M. Rader, “The chirp-z transform algorithm and its applications,” *Bell System Tech. J.*, vol. 48, pp. 1249-1292, 1969.

[6] W. H. Press, S. A. Teukolski, W. T. Vetterling and B. P. Flannery, *Numerical Recipes in C, The Art of Scientific Computing*, Cambridge Univ. Press, 2002.

[7] S. A. Khan, *Digital Design of Signal Processing Systems: A Practical Approach*, Wiley, Section 2.5.4.4, 2011.

[8] D. Cohen, “Simplified control of FFT hardware,” *IEEE Trans. Acoustics, Speech, Sig. Proc.*, pp. 577-579, Dec. 1976.

[9] CORDIC - http://en.wikipedia.org/wiki/CORDIC

[10] Gnu Octave - http://www.gnu.org/software/octave/