Fast Fourier Transform Implementation for High Speed Astrophysics Applications on FPGAs

Tomasz S. Czajkowski, Christopher J. Comis, Mohamed Kawokgy
Edward S. Rogers Department of Electrical and Computer Engineering
University of Toronto, 10 King's College Road, Toronto, Ontario, M5S 3G4, Canada
{czajkow|comis}@eecg.toronto.edu, kawokgy@vrg.utoronto.ca

Abstract

Research in astrophysics requires high-speed logic circuits to perform real time signal processing. A common algorithm used in signal processing is the Fast Fourier Transform (FFT). In this research we have created two distinct FFT implementations targeting small to medium size Altera Stratix FPGAs. The first implementation focuses on lowering the number of used built-in multipliers by using only a single complex multiplier in each stage of the algorithm. The second approach shows an implementation of the algorithm to lower the latency of the circuit by using several multipliers in parallel in each stage of the FFT to speed up computation. While both approaches work, the designer may want to decide which implementation to use based on costs associated with obtaining a device that can facilitate each of FFT algorithms implementations.

We successfully implemented the architecture with a single complex multiplier per stage, as this architecture was feasible given our design constraints. We discuss the methodology used to create this design. Also the results in terms of speed and resource utilization of the design are discussed.

1. Introduction

One of the most important contributions of Astrophysical research is the discovery of when major events in the Universe took place. One of such discoveries relates to the moment in the life of the Universe when it became ionized. To determine various facts about the Universe, the electromagnetic radiations received on Earth from objects moving through space, such as Hydrogen I, or HI, particles as they travels away from Earth, need to be examined.

It is know that stationary HI emits electromagnetic radiation with the frequency of approximately 1.4GHz [1]. However, as the HI moves through space the frequency of received radiation is altered due to the Doppler Effect, also known as Doppler Shift [1]. As HI moves away from the earth, it is expected that the frequency of its radiation to be reduced, or red-shifted, to somewhere between 70MHz and 200MHz, as a result of the Doppler Effect. To determine the frequency of radiation, we need to sample the waveform received from atmosphere, at least at twice its maximum frequency, to satisfy the Nyquist condition and avoid aliasing [2]. That is, at a rate of 400MHz, or 400MSamples/s.

To analyze the frequency content of the radiation received from the cosmos we employ a tool known as the Fourier Transform. The Fourier Transform is a mathematical tool that maps a signal from a time domain to an equivalent representation in terms of sinusoidal waveforms of varying frequencies and amplitudes. We can plot the amplitude versus frequency for each result obtained from the transform to obtain a visual representation of how strongly a specific sinusoid frequency affects the input signal. In the case of a search for the frequency of radiation of HI, astrophysicists attempt to distinguish the frequency content of the background noise from the cosmos from the frequency of radiation of HI.

The Fourier Transform however is a time consuming computation. It is of interest to process the data in real time, rather than wait significant amount of time for processing old data. By using an algorithm known as the Fast Fourier Transform it is possible to implement a real time signal processing system.

The focus of this project is to use an Altera Stratix S20 FPGA to perform real-time signal processing on incoming atmospheric Electromagnetic Radiation. Using this data, we can specifically analyze the HI electromagnetic radiations to determine its precise frequency. In turn, this data can provide us with insight into when the Universe turned from being neutral to being ionized. Figure 1 shows a system-level setup of this prototypical project.

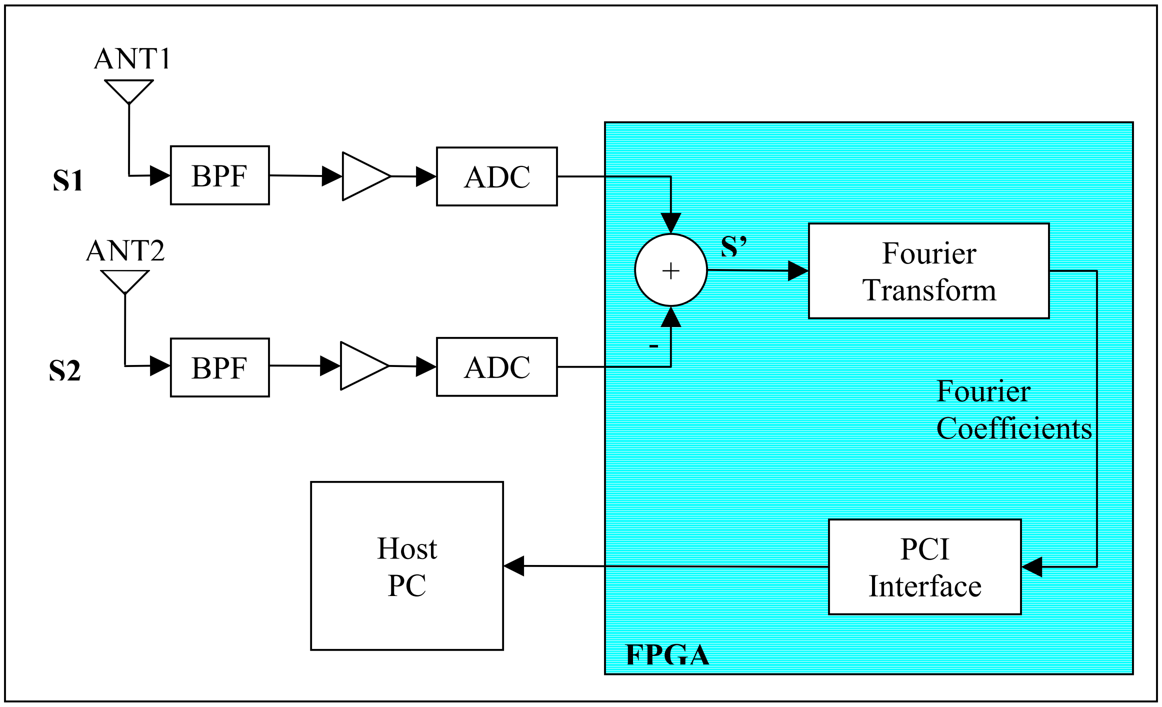

Figure 1: System-Level Project Setup

The input to the design comes through a pair of antennae, ANT1 and ANT2, which are exposed to the ambient atmosphere. The received signals S1 and S2 are amplified and Band-Pass Filtered to limit the spectrum of the signal to the frequency range of 70 MHz to 200MHz. Both input signals pass though an Analog to Digital converter (ADC), where they are sampled at 200MHz, resulting in an 8-bit precision output representing the amplitude of the input signal. By taking the difference between the two sampled data streams, any noise components from the waveform can be filtered out. A real-time Fourier Transform is performed on the samples received. The Fourier Transform of the samples gives an idea about the frequency content of the received signal and, thus, the frequency of oscillation or radiation of the ionized HI. The resulting Fourier coefficients from the Fourier transform are then stored and sent though the PCI interface to the host PC for further processing.

1.1 Specifications

The primary challenge in this project is to implement a Fast Fourier Transform (FFT) algorithm such that it can process data at a rate of 200 MSamples/sec. In addition, the FFT frame size to be processed is 32K samples, which makes this operation computationally intensive, even if the algorithm is O(NlogN). The target device is the Altera Stratix S20, which represents another challenge in the implementation of such specifications because of the hardware limitations.

1.2 Methodology

Figure 2 shows the flow of the methodology used to transform the application of interest from an abstract idea into actual hardware. First, the application of interest is well studied and literature search is performed to understand what has been done before and to determine the main challenges to be faced. Second, given the specifications provided, a high level model is analyzed using Matlab to ensure that such specifications are suitable for the task and to gain more understanding of the effect of the different variables, such as the number of points used by FFT as well as the input data sampling rate. Then a more detailed Matlab fixed point model is designed to decide on the word lengths to be used for different sections of the design as well as the effect of quantization error on the final performance. Then VHDL modelling for each part of the design is performed based on a divide-and-conquer type of strategy. Basic functional and timing simulations are then performed, using Quartus, to ensure that the design is meeting the functional as well as the timing specifications set. Then, more elaborate simulations are performed using Testbuilder. Finally, the design is downloaded to the Altera Stratix S20 device for real-time hardware testing.

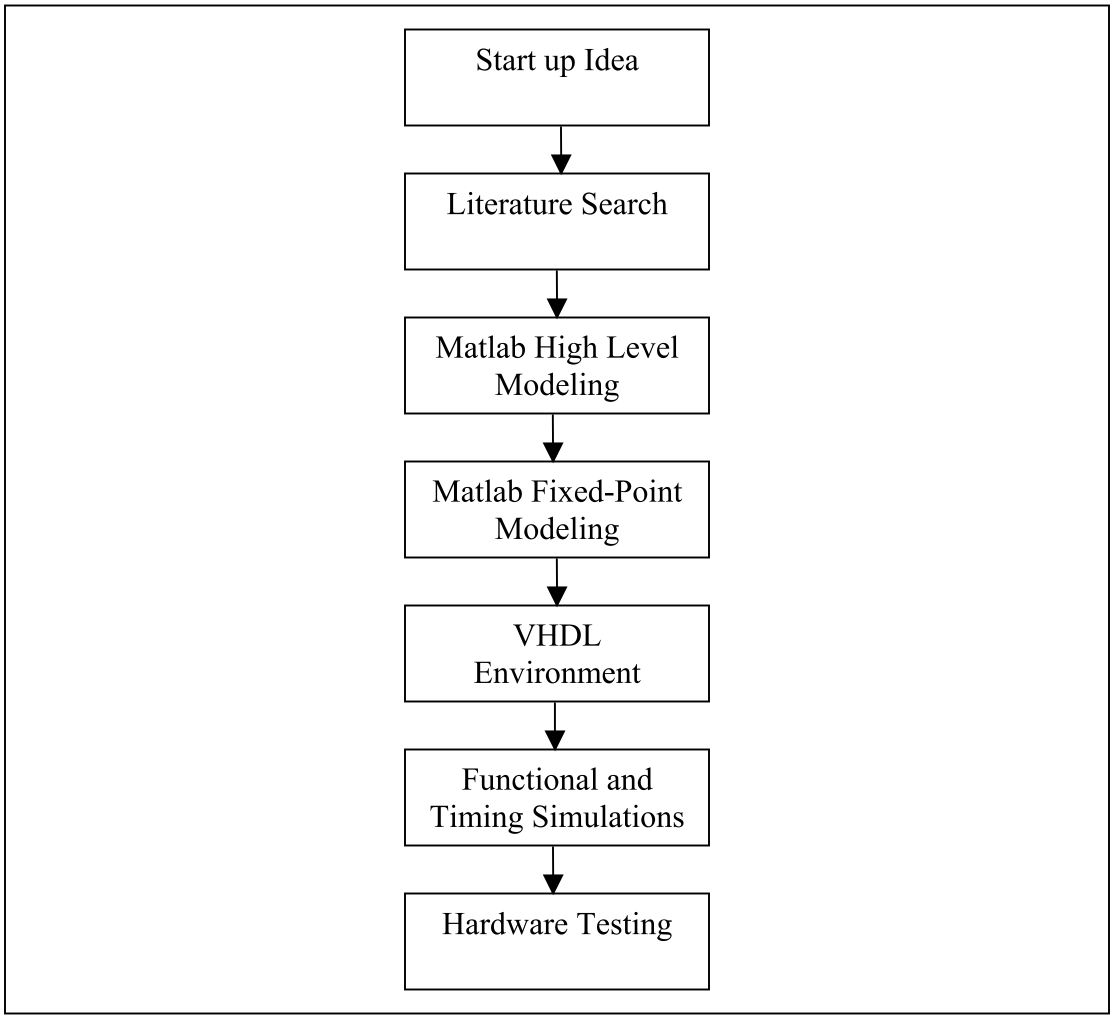
Start up Idea
Literature Search
Matlab High Level Modeling
Matlab Fixed-Point Modeling
VHDL Environment
Functional and Timing Simulations
Hardware Testing

Figure 2: Design Methodology

1.3 Current Status

The Fast Fourier Transform algorithm has been successfully implemented on the Altera Stratix 20k device. We have been able to improve the operating speed of the circuit beyond that available in commercial FFT cores for the Altera Stratix device. The design has been fully tested within a hardware testing framework to verify that the software simulations and the hardware behaviour of the design indeed match. While only a 16384 point FFT with 18-bit precision was implemented on the Altera Stratix 20k device, it is possible to extend this design to include more points and data precision on larger devices.

1.4 Organization

This report is organized as follows. In Section 2, background information, necessary to understand the different parts of the design, is presented. Section 3 describes the implemented design from an architectural point of view. Different architecture options are presented along with Matlab high-level modelling and simulation results with emphasis on the architecture of choice. Section 4 discusses the actual hardware implementation of the different building blocks of the design with detailed design issues and considerations. Section 5 discusses integration of the FFT into a top-level embedded system. Testing methodology including functional simulations, Testbuilder simulations, and hardware testing is presented in Section 6. Results are discussed in Section 7. Contributions of different group members followed by Conclusions are presented in Sections 8 and 9, respectively.

2. *Background*

This section is organized as follows. First, the concept of spectral analysis using Discrete Fourier Transform is presented with emphasis on its computational non-efficiency. Second, a more computationally efficient algorithm, namely the Fast Fourier Transform (FFT), is discussed and proved to be a more attractive option to implement.

2.1 The Fourier Transform

The Fourier Transform is the mathematical entity that is used to perform spectral analysis on an arbitrary signal in order to get information about its frequency contents or components. When the signal of interest is a discrete signal, consisting of discrete time samples, the Fourier Transform is referred to as Discrete Fourier Transform (DFT) [2]. Given a data sequence x[n] of length N, the computational problem of the DFT is to compute the sequence $X(e^{j\omega})$, of N complex-valued numbers, representing the spectrum of the input data sequence. Since $\omega=2\pi k/N$, we abbreviate the notation $X(e^{j\omega})$ to X(k) and compute the DFT according to the formula in Equation (1):

$$X(k)=\sum x[n]W_N^{nk}, \qquad 0 \leq n,k \leq N-1 \qquad (1)$$

In Equation (1), W_N is equal to $e^{-j2\pi/N}$, which represents a specific point in the complex domain. For the remainder of this document, this complex exponential will be referred to as a Twiddle Factor.

To compute the Discrete Fourier Transform (DFT) of a given N-point data sequence x[n], Equation (1) suggests that each sample of the sequence x[n] must be multiplied by the corresponding Twiddle Factor W_N^{nk}. Thus, for N-point DFT, N Twiddle Factors are to be stored and multiplied by the corresponding data sample of the N-point data sequence. Since each Twiddle Factor is basically a complex exponential, the N Twiddle Factors for an N-point DFT can be represented in a complex plane evenly distributed along the unit circle periphery. Figure 3 shows the complex plane with sufficient Twiddle Factors for the 8-point DFT case.

In general, the data sequence x[n] is also assumed to be complex valued. We observe that for each value of *k*, the frequency index, direct computation of X(k) involves N complex

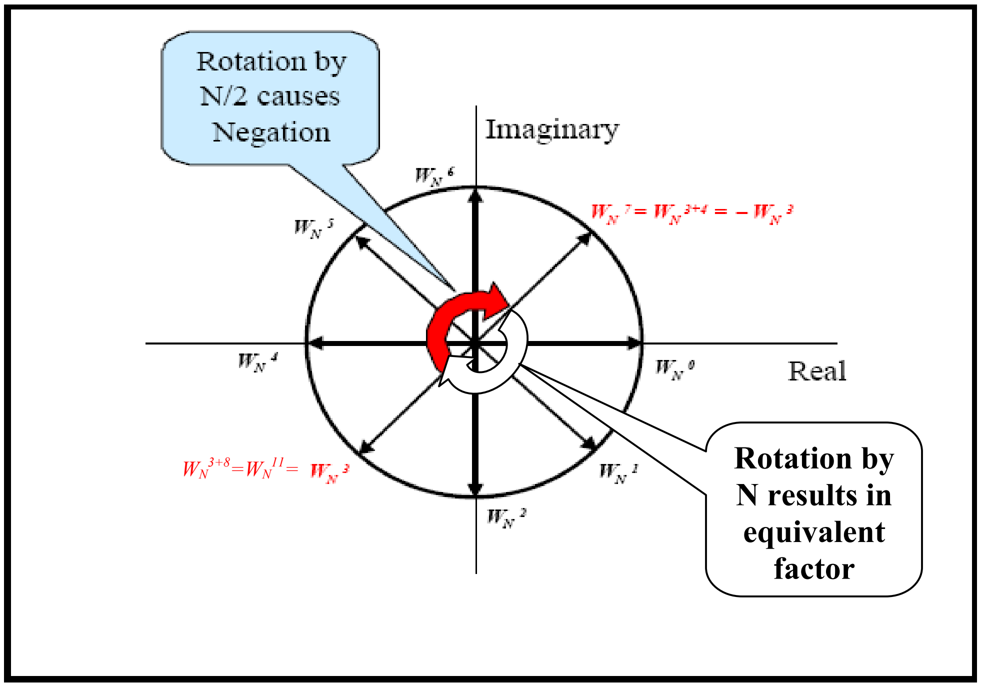
Rotation by N/2 causes Negation
Imaginary
W_N^6
$W_N^7 = W_N^{3+4} = -W_N^3$
W_N^5
W_N^4
W_N^0
Real
$W_N^{3+8} = W_N^{11} = W_N^3$
W_N^1
W_N^2
Rotation by N results in equivalent factor

Figure 3: Twiddle Factors Complex Plane for 8-Point DFT

multiplications (or equivalently 4N real multiplications) and N-1 complex additions (or equivalently 4N-2 real additions). Consequently, N-point DFT computation requires N^2 complex multiplications and N^2-N complex additions. A direct computation of the DFT is inefficient, because it doesn't exploit the symmetry and periodicity properties of the Twiddle Factor W_N. Referring to Figure 3, these two properties are:

$$\text{Symmetry property: } W_N^{k+N/2} = -W_N^k$$
$$\text{Periodicity property: } W_N^{k+N} = W_N^k$$

An improved approach that utilizes these properties to reduce the computation time of the Discrete Fourier Transform is known as the Fast Fourier Transform algorithm.

2.2 The Fast Fourier Transform

The Fast Fourier Transform takes advantage of the symmetry and periodicity properties of the Fourier Transform to reduce computation time. It was noticed that when the number of points, N, used by the FFT computation was a power of 2 then it is possible to divide the complex multiplication into $\log_2(N)$ stages and generate all required complex exponential multiples of input values by performing computation on a pair of input values at a time. Such design of an FFT is referred to as a radix-2 FFT design. To optimize performance of the FFT circuit designs that process 4 or 8 elements at a time were designed (known as radix-4 and radix-8) which require the value of N to be a power

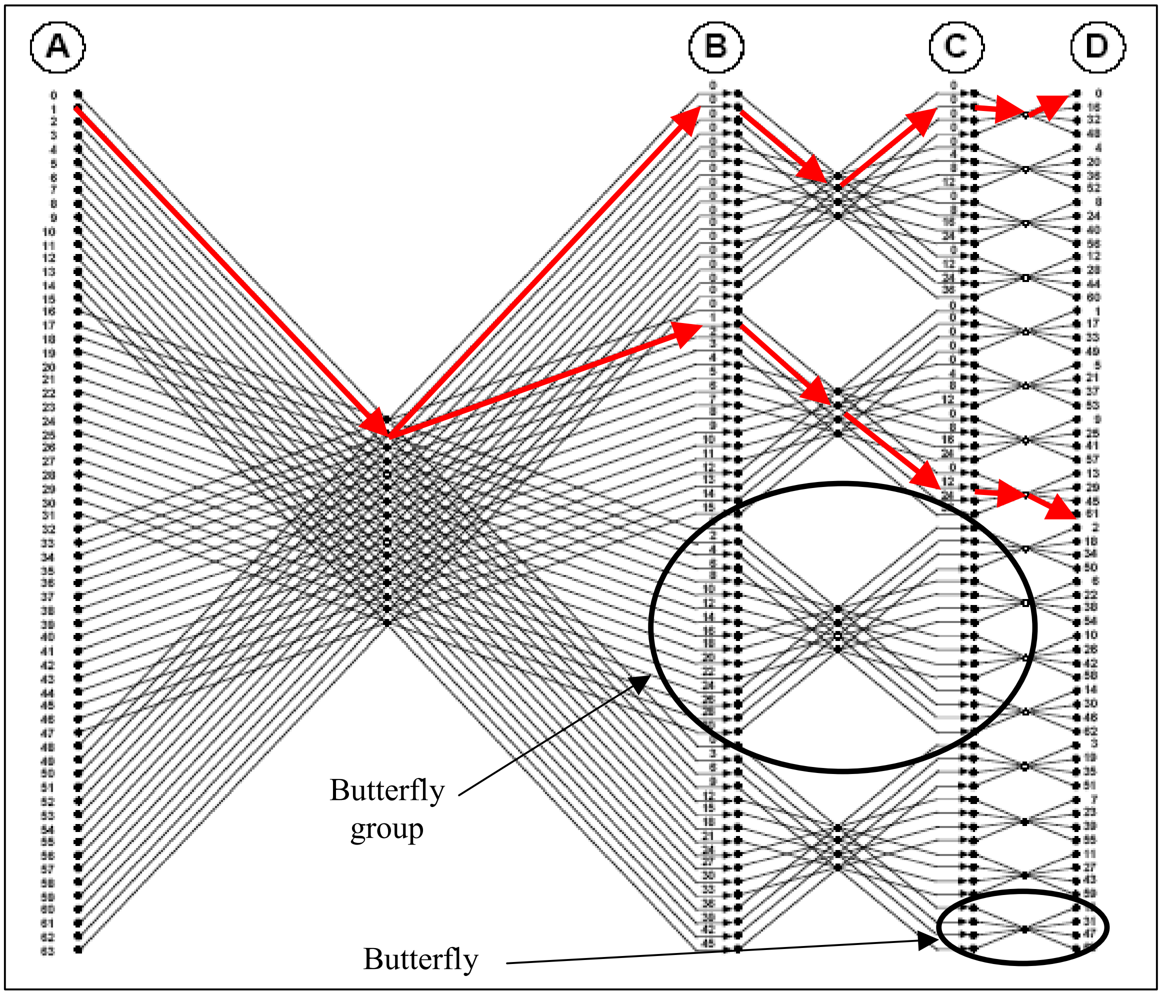

Figure 4: 64-Point FFT Example

of 4 and 8 respectively. The most popular design is the radix-4. An example of this design is shown in Figure 4.

In Figure 4 we compute a 64-point FFT. Each 4-input/4-output computational unit, known as a butterfly, takes a weighted sum of inputs and produces results with different weights at every output. Each output k, indexed 0 through 3 from top to bottom, is then multiplied by a Twiddle Factor W_N^{nk}, where n is a zero-based index of the butterfly within a butterfly group with 0 being the topmost butterfly in the group.

As an example consider the complex exponential multiples of x[1] for output values X(0) and X(61). For X(0) the summation affecting x[1] uses the Twiddle Factor W_N^0 for the final result. By tracing a path from x[1] to X(0) (top highlighted path) we notice that the Twiddle Factor values, numbers over the horizontal arrow in Figure 4, add up to 0. For the lower highlighted path the Twiddle Factor exponents should be W_N^{61} (W_N^{n*k}), but it is only W_N^{13}. The remaining factor of W_N^{48} is hidden away in the design of the computational unit, also known as the butterfly.

The butterfly is a combination of adders and multipliers that manipulate the data passing from inputs to outputs. For the example in Figure 4, the butterfly unit is shown in Figure 5. We notice that in the first segment, the input x[1] attaches to the input x[1] of the butterfly and passes to the output y[1]. In Figure 5 we notice a multiplication by –j, which is equivalent to multiplication by Twiddle Factor W_N^{16}. In the second stage we see that the intermediate value based on x[1] comes in as input x[0] to the butterfly and is output through output y[3], while being multiplied by 1. Finally, in the last butterfly the data passes from input x[1] to output y[3] while multiplying the data by j, which is equivalent to multiplying by a Twiddle Factor W_N^{32}. Thus, within the butterfly modules the input x[1] is multiplied by a Twiddle Factor $W_N^{16} * W_N^{32} = W_N^{48}$. Thus, the final complex exponential used to multiply the input x[1] is as it should be.

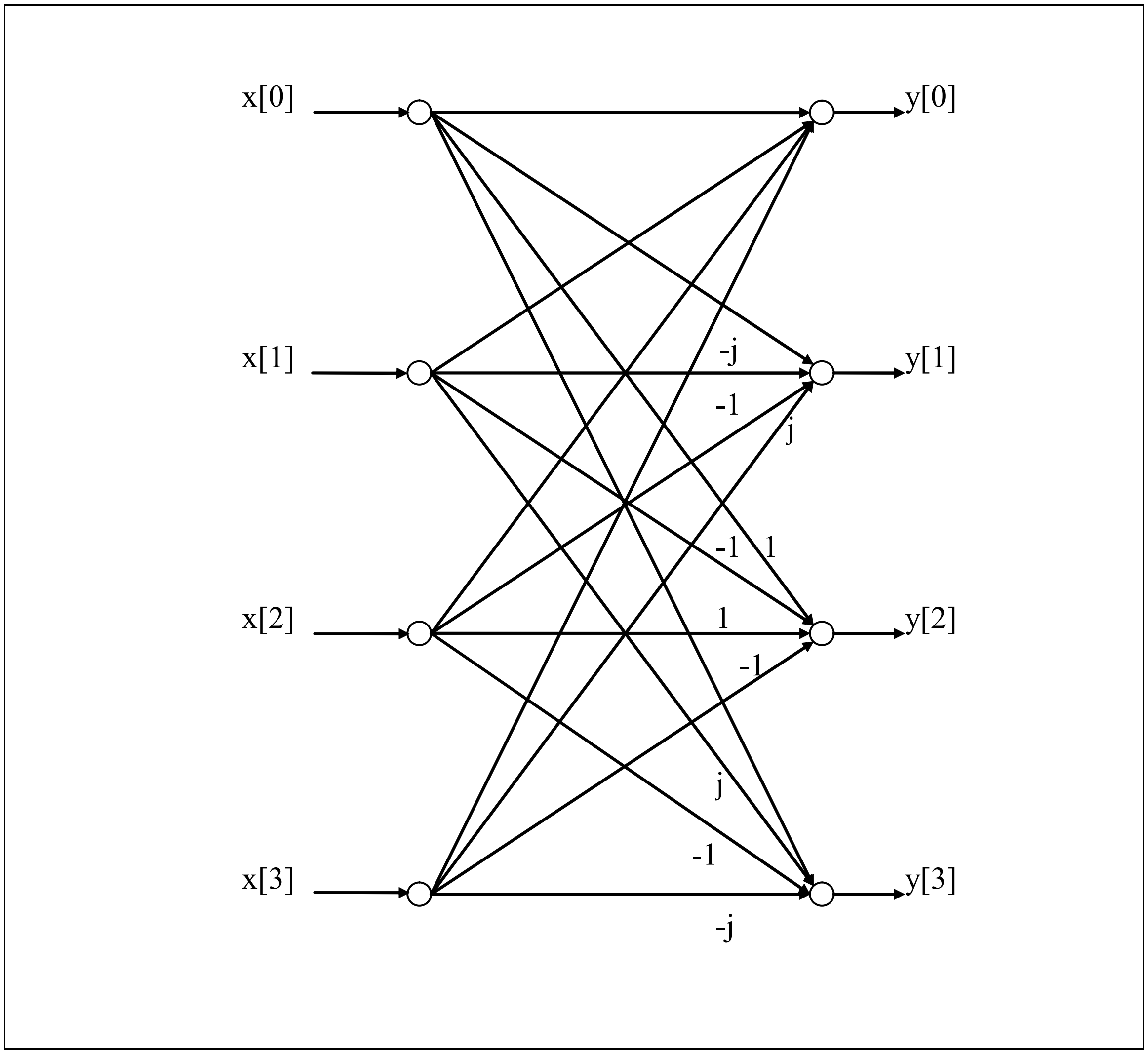

Figure 5: A butterfly without Twiddle Factor multipliers

2.3 Altera Stratix FPGA

The above algorithm can be implemented on a Field-Programmable Gate Array device, such as an Altera Stratix device. An FPGA is an array of logic blocks that can implement logic functions interconnected by a programmable routing network. The main advantage of FPGAs is that custom computational units, such as the butterfly units, can easily be implemented in parallel. In addition to reprogrammable blocks, called Lookup Tables (LUTs), which can implement any function of 4 boolean variables, the Stratix FPGA has memory blocks to store data and built in Digital Signal Processing (DSP) blocks [3].

A DSP block consists of hardware multipliers and adders that operate much faster than its equivalents constructed using LUTs. The Altera Stratix DSP multiplier units are optimized for 9-bit multiplication, but can be combined together to multiply wider inputs. In later sections we show how we utilized these multipliers in our design to speed up the circuit.

3. Architectural Design

The first step in designing a hardware system is to create a model that accurately represents the design. By using this model the designer can manipulate parameters and specifications to achieve reasonable results while meeting the design specifications. In our case the model represents a section of the FFT, also known as a butterfly.

In this section of the report we specify parameters that need to be considered to implement a design that produces reasonable results. We then discuss two Fast Fourier Transform butterfly architectures that could meet our design constraints and a method generate complex exponentials, also known as Twiddle Factors, which is used to accommodate the needs of the second butterfly architecture. To observe the operation of both architectures we developed Matlab models for them and performed simulations to compare floating point FFT results generated by Matlab fft() function to the output generated by our models. These results were crucial in determining operating parameters for the final implementation of the FFT in hardware.

3.1 Design Considerations

We begin our analysis by specifying design parameters that need to be considered before implementing our circuit in hardware. These parameters are:

1. precision of data – *bit-width*,
2. number of points used for computation – N,
3. memory usage – M, and
4. desired circuit operating frequency – f_{op}

The precision of data relates to the number of bits per number used by the butterfly, we refer to as *bit-width*. Both real and imaginary data values will be composed of the number of bits specified by this parameter. The bit-width parameter has a direct impact on the quality of results as the fewer bits we use the less precise the result will be. For example, with the bit-width parameter set to 4 each number will only be able to take 16 distinct values. This may not be sufficient to represent the data and will introduce round-off error. Thus, we want to keep this parameter as high as possible. However, a practical consideration is the number of built in multipliers that the target FPGA device has. As discussed in section 2.3, the Altera Stratix 20k device has 80 9-bit multipliers. This imposes an upper bound on the bit-width parameter as we require all multipliers to be composed of the build-in 9-bit multipliers. While it is possible to use regular logic to implement multipliers as well, multipliers composed of lookup tables are considerably slower and larger. If more multipliers are needed then the end user will be required to use a larger Altera Stratix device. Thus, a proper choice of the value of bit-width is crucial as it will also impact the Altera Stratix device we will need to use to implement the design.

The second parameter is the number of points used for computation, N. As discussed in section 2.1, increasing the number of points used for a Discrete Fourier Transform improves the quality of the results, allowing us to analyze more complex waveforms. While we want to maximize this parameter we have to keep in mind that each point we use increases the memory requirement of our design by 2*(bit-width), since both real and imaginary parts of a point need to be saved. Thus, the upper limit on the number of points is bounded by the size of memory available on the device. Thus, the choice of N directly affects the value of M that has to be less than the total memory available on the FPGA device.

The final parameter is the circuit operating frequency, f_{op}. This parameter defines the slowest speed the circuit can work at and still be able to process the incoming data without data loss. This depends entirely on the number of points the FFT butterfly can process at any given time. Thus, in our design we will be looking to minimize this value, as it will make it easier to implement on the FPGA. If we manage to achieve an operating frequency that is higher than f_{op} then we will be able to speed up the data input rate, which provides the end-user with the option to increase the data sampling rate that in turn improves the quality of their results.

In the following sections we discuss two FFT butterfly architectures and address the parameters discussed above. We will then use those parameters to develop hardware models in section 4 and decide on which butterfly to use for our final circuit implementation.

3.2 Butterfly Architecture A

The first architecture we present is a modification on the basic butterfly architecture described in section 2, and maybe seen in Figure 6. In most implementations after each multiplication the result is extended by 2 or 3 bits, or right shifted by 2 or 3 bits. While this is done to make sure that the most significant bits of the result do not get lost and to prevent overflow, some data precision is lost. This penalty is not always necessary. In this section we describe a simple, but effective, method of retaining more precision during computation while still preventing the overflow from occurring.

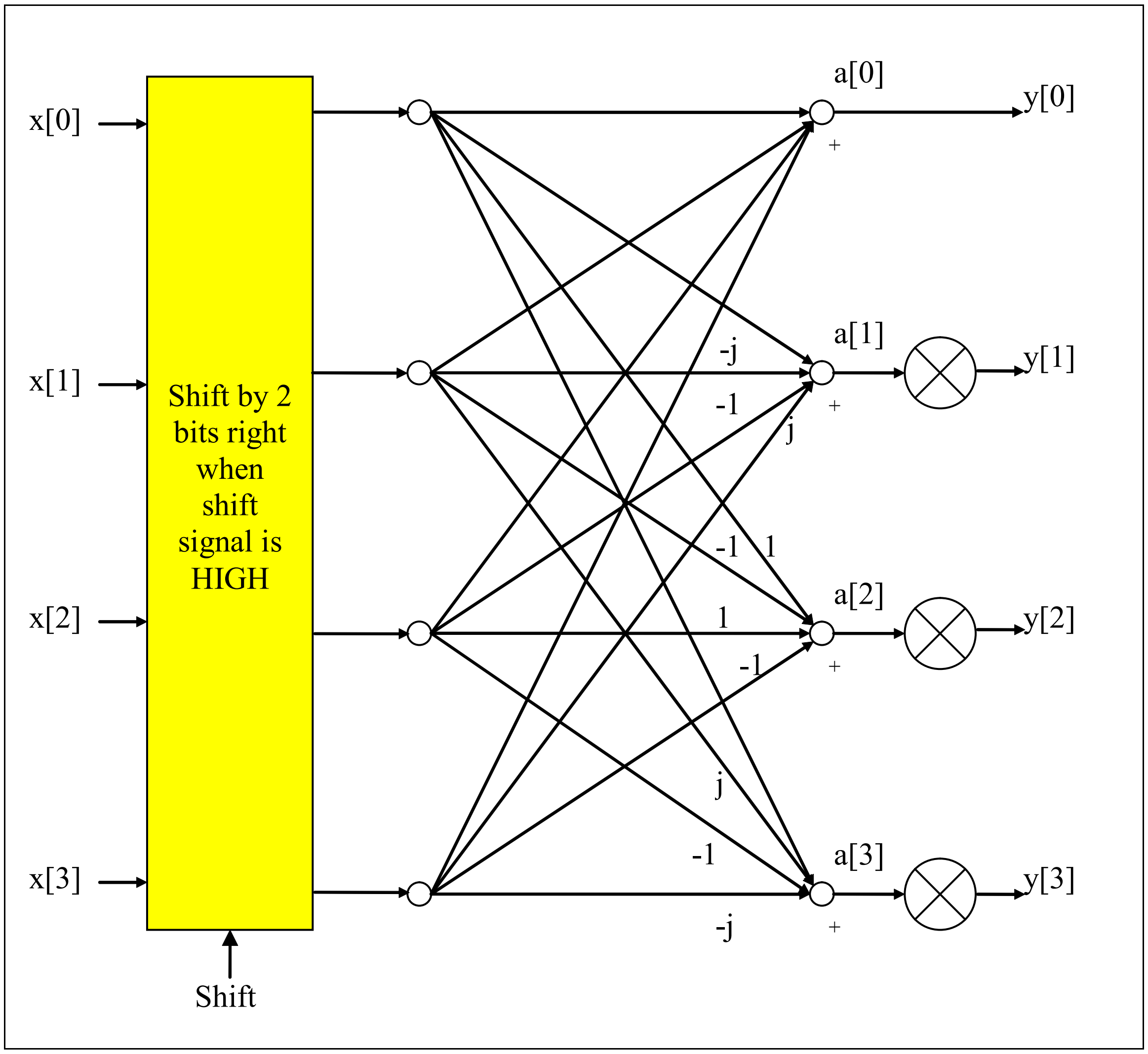

Figure 6: Architecture A Butterfly Design With Shift-Only-When-Necessary Logic

We notice that it is not always necessary to shift the data to the right, but only when it can cause overflow. Thus, we examine the data produced by each stage of the FFT computation and decide if a shift is necessary. According to [4], a shift of 3 bits after the first stage is required to prevent overflow, while in the other stages only a shift of 2 bits is necessary. This means that for all stages, except the first one, we have to make sure that the 3 most significant bits of either the real or the imaginary component are equal. If they are not there is a possibility of overflow. Thus, if this case is detected we shift the inputs coming to the next stage by 2 bits to the right.

In this architecture we design a single FFT stage that performs computation for all stages of the algorithm. In this single stage we implement several radix-4 butterflies, shown in Figure 6, to speed up computation. Since there cannot be enough butterfly units to compute all points simultaneously, we adopt a scheme where we shift if any of the butterflies used detects a possibility of an overflow on any of its outputs.

Index	Re(x)	Im(x)	Re(a)	Im(a)	Re(y)	Im(y)	Need Shift?
0	00000001	00000000	00001010	00000000	00001010	00000000	No
1	00000010	00000000	11111110	00000010	11111110	00000010	No
2	00000011	00000000	11111110	00000000	11111110	00000000	No
3	00000100	00000000	11111110	11111110	11111110	11111110	No

Table 1: Processing 4-point FFT using butterfly architecture A

To store intermediate data and input points we use the same memory, as once a point is used for computation, it is never needed again. To cope with real-time incoming data, we use a secondary buffer to store incoming data while the FFT circuit is computing.

3.2.1 Computing the FFT

Let us examine how a 4-point FFT is computed using this butterfly architecture in Figure 6. As an example, suppose that the bit-width parameter is set to 8, and consider the inputs 1+0j, 2+0j, 3+0j and 4+0j. First, the data is passed to the four inputs of the butterfly, namely x[0], x[1], x[2], and x[3]. Since this is the first stage there is no shift to the right and the data passes to the two series of adder units. The last series of adders produce values a[0], a[1], a[2], and a[3]. In turn these results pass to the complex multipliers. In this case the complex multipliers multiply by 1+0j, resulting in values y[0], y[1], y[2], y[3]. To check if a shift is necessary in the next stage we look at the top 3 bits of y[0], y[1], y[2] and y[3]. Since top 3 bits of each number are the same then shift is unnecessary. The resulting values of y[…] in decimal are 10, -2+2j, -2, -2-2j respectively. These results are summarized in Table 1.

3.2.2 Output Ordering

The last thing to note is the order of outputs produced by the radix-4 butterfly shown in Figure 6. While in the example the input and output ordering is identical, if more points in are included in the FFT we notice that the results are output in the digit reverse order [2].

The digit reverse order refers to reading the index of the output value treating the right-most digits as most significant, versus the usual convention of left digits being most significant. For example, in base 10 representation the value 1234, the digits reverse number would be 4321. We apply the same procedure to decode the index of the result using base 4 digits. Thus, starting from the top of the butterfly for say 16 points the first output index would be $(00)_4$, then $(01)_4$, $(02)_4$ and $(03)_4$. By applying digit reversal we obtain indices $(00)_4$, $(10)_4$, $(20)_4$ and $(30)_4$, which in decimal correspond to 0, 4, 8 and 12. This procedure is well known and documented.

3.2.3 Addressing Design Considerations

We now consider how the butterfly architecture A affects the four parameters described in section 3.1.

The first parameter to consider is the *bit-width*. While the more bits are used the better the result, the main concern here is availability of multipliers on the FPGA. As you can see each butterfly unit uses 3 complex multipliers, which translates to 12 real multipliers. Therefore, the feasibility of this architecture heavily depends on the number of available multipliers on the FPGA. On the other hand, we will show in section 3.5.1 that the shift-only-when-necessary strategy can reduce the required bit-width value in the case of signals that have relatively flat spectra.

In terms of the number of points, *N*, we can use we note that with this architecture we reuse the same memory space for intermediate data. This requires us however to use a second buffer to store data that arrives while the FFT unit is processing data, so we should not use more than half of the memory available on the FPGA device. This restricts the parameter M to less than half of all available memory on the FPGA.

On the positive side, the parallel nature of this architecture lowers the f_{op} requirement for the design. Notice that at each given time a single butterfly processes 4 points simultaneously. Thus, for 16384 points if we use 7 radix-4 butterfly units then the required f_{op} will be reduced to mere 200MHz / 4 = 50MHz. This is a major advantage, as it is very easy to achieve this speed on current FPGA devices. Achieving a higher speed than 50MHz will provide us with some flexibility on how many butterfly units will be used in the design.

Overall, this architecture is promising and requires a very low f_{op}. The only concerns are the availability of on-chip RAM and hardware multipliers on the FPGA, as for high precision computations a chip larger than Stratix 20k may be required.

3.3 Architecture B

The second architecture we discuss is based on a design by Despain, 1974 [1]. This architecture was conceived to minimize the number of multiplier units in the circuit. At the time multiplication was a very expensive operation, thus reducing the number of multipliers in a design was a good way to improve the design. While the multipliers currently on FPGAs are much faster than in the past, the desire to lower the number of multipliers in the design is now driven by the need to reduce costs of an FFT design and the ability to fit the design on a smaller FPGA. On smaller FPGA devices the number of multipliers is not sufficient to facilitate FFT algorithms with more than 1 or 2 multipliers per stage[1], this design is again very attractive. In this section we present an architecture described by Despain [5] that uses only one complex multiplier per FFT stage.

[1] This depends on number of points, *N*, and data precision, *bit-width*. For example, at 18-bit precision 8 multiplier units are used per single complex multiplier on an Altera Stratix device. On a Stratix 10K device there are only 40 such multiplier units, thus if we use an architecture with 3 complex multipliers per stage we can have only one stage of the FFT in the design (4 points). On the other hand, if only 1 complex multiplier per stage is used we can implement up to 5 stages of FFT, or 1024 point FFT.

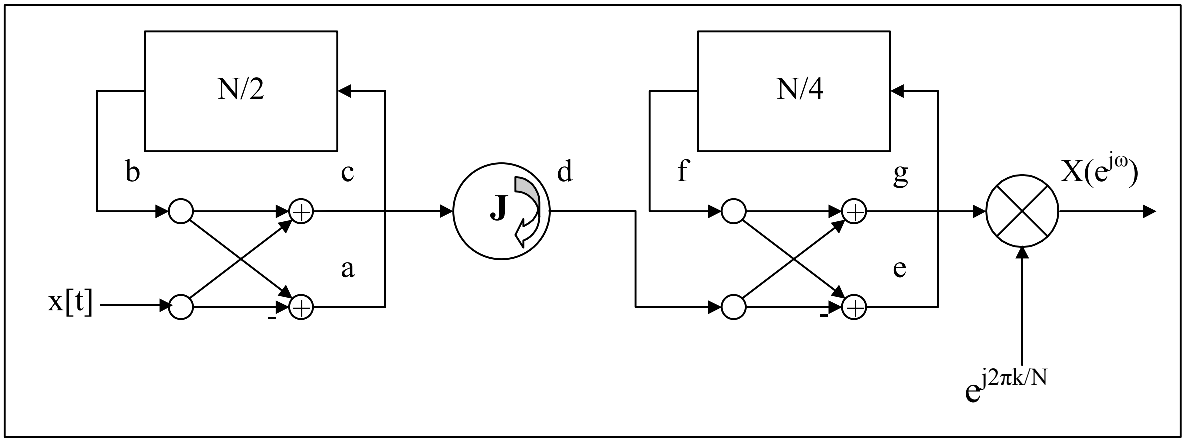

Figure 7: Radix-4 Butterfly Architecture B with a Single Multiplier Unit

In this architecture we use $\log_4(N)$ serially connected radix-4 butterflies to compute the FFT. Unlike the previous architecture the same memory space is not used to store points and intermediate data as the data flows from one stage to the next, never backtracking. This eliminates the need for an input buffer as in architecture A and also changes how the butterfly behaves. In section 3.2 we discussed the operation of the Fast Fourier Transform algorithm based on commonly used radix-4 butterfly architecture. We use this as a reference to describe how the radix-4 butterfly described by Despain and Wold [5][6] computes the transform. We first talk about the general computation performed by the butterfly and then discuss the order in which the results appear at the output of the circuit.

3.3.1 Computing the FFT

The butterfly architecture described by Despain and Wold [5][6] is shown in Figure 7. The main difference between this architecture, we refer to as architecture B, and the previous architecture is that the computation is performed in a serial manner. In the butterfly there are memory units that function like shift registers that delays the incoming data to allow each output of a radix-4 butterfly to be computed serially. We explain how it works using a 4-point FFT as an example, just like in section 3.2.1.

Time	x[t]	a	Left Shift Register	b	c	d	e	Right Shift Register	f	g
0	1+0j	1+0j	0, 0	0	0	0	0	0	0	0
1	2+0j	2+0j	0, 1+0j	0	0	0	0	0	0	0
2	3+0j	-2+0j	1+0j, 2+0j	1+0j	4+0j	4+0j	4+0j	0	0	0
3	4+0j	-2+0j	2+0j, -2+0j	2+0j	6+0j	6+0j	-2+0j	4+0j	4+0j	**10+0j**
4	0	0	-2+0j, -2+0j	-2+0j	-2+0j	0-2j	0-2j	-2+0j	-2+0j	**-2+0j**
5	0	0	-2+0j, 0	-2+0j	-2+0j	-2+0j	-2+2j	0-2j	0-2j	**-2-2j**
6	0	0	0, 0	0	0	0	0	-2+2j	-2+2j	**-2+2j**

Table 2: Data flow through the butterfly architecture B

Consider the same input as in section 3.2.1, namely 1+0j, 2+0j, 3+0j and 4+0j. First, the inputs pass through a radix-2 butterfly. This butterfly either passes the value horizontally or performs a sum of inputs *b* and *x[t]* for the output *c* and a difference of *b* and *x[t]* for output *a*. For the first N/2 clock cycles the butterfly passes the values without change, resulting in 1+0j and 2+0j being stored in the left shift register. At that point the butterfly switches to sum and difference computation. With the input x[t] set to 3+0j, the output *c* becomes 4+0j and output *a* becomes -2+0j. The output *c* now passes through a J-multiplier, which multiplies values passing through it by **j** during clock cycles [0, N/4) when the radix-2 butterfly does not perform sum-difference computation.

The value at *c* thus passes to *d* and is processed by the second radix-2 butterfly. The operation of the second radix-2 butterfly is the same as for the first radix-2 butterfly, but it switches to sum and difference computation every N/4 clock cycles. The first value that arrives at the second radix-2 butterfly initiates its operation. Thus the first value *d* is passed to *e* and is stored in the right shift register.

At the next clock cycle the values of *x[t]*, *b*, *d* and *f* are 4+0j, 2+0j, 6+0j and 4+0j respectively. The value computed for node *a* is -2+0j, causing the left shift register to contain -2+0j, -2+0j values from this and the last computation. The value *c* is computed to be 6+0j and is again passed to *d*. Now that the second radix-2 butterfly is in the sum and difference mode *g* becomes 10+0j and *e* is set to -2+0j. Since for 4 point FFT the complex exponential is set to 1+0j, output *g* becomes the final output of the butterfly.

In the next clock cycles the J-multiplier multiplies incoming data *c* by **j**, while the first radix-2 butterfly is passing data from *b* to *c* and the second radix-2 butterfly. Thus, the output *g* becomes -2+0j and value 0-2j is stored into the right shift register. In the following clock cycles the final value is read from the first shift register (-2+0j) and passed through the J-multiplier to node *d*. Now that the second radix-2 butterfly is producing sum and difference of its inputs, the output *g* becomes -2-2j is output by the butterfly, while the value of *e* (-2+2j) is stored in the right shift register.

In the last clock cycle the value in the right shift register is passed to the output of the butterfly, completing the computation. This computation is summarized in Table 2.

3.3.2 Output Ordering

When we compare the results of the computation in section 3.3.1 to those in section 3.2.1 we notice that while the results are the same, the ordering in which the results are produced is different. While in most algorithms the butterfly produces outputs in a digit-reversed order, this design is different. We notice that the first two results produces are for even indices (0 and 2) while latter two results are for the odd indices (1 and 3).

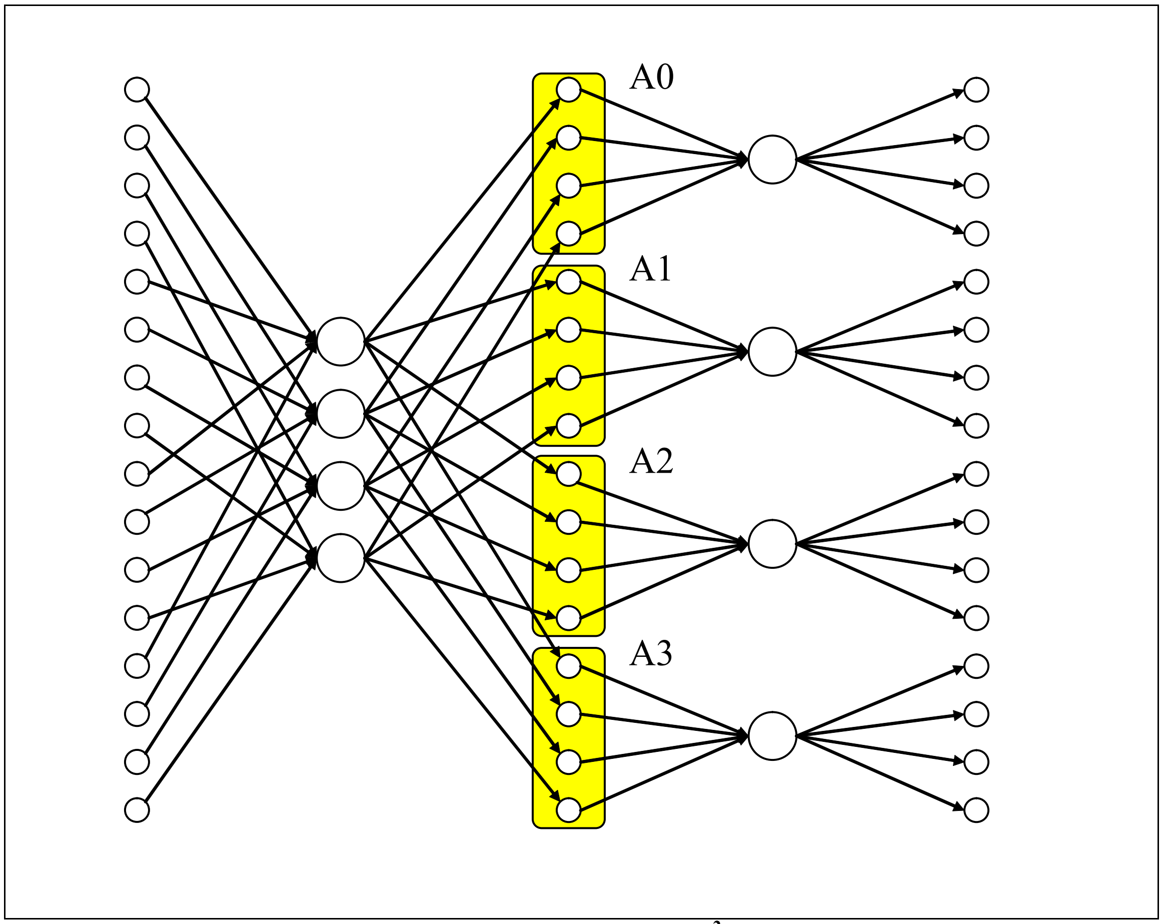

Figure 8: 16 point FFT [2]

By referring to Figure 8, we continue our analysis to discover what effect this has on the final ordering of the data when the Fourier Transform uses 4^s points, where s>1. We notice that for s=2, this butterfly architecture first computes partial results in group A0, then A2, A1 and A3. Thus, to put the results in the digit reverse order, we have to swap groups A1 and A2 and then swap entries 1 and 2 within each group. This suggests a recursive algorithm. In this algorithm we take all *N* outputs of the FFT, divide it into 4 equal sections and swap the middle two sections. The recursion comes in when we call the same procedure on each of the four sections, until the size of each section is 1. The same procedure can be implemented iteratively, as shown in Figure 9.

[2] Groups A0 through A3 are groups of results generated by the 0th, 1st, 2nd and 3rd leg of the butterfly. The omitted exponent values correspond to those presented in section 2

```
Function ReorderData(data, num_points)
{
    int num_stages = log4(num_points);

    for (index = 4; index <= num_points; index = index + 4)
    {
        step = 1;
        for (stage = 1; stage <= num_stages; stage++)
        {
            if  index mod (step*4) == 0
            {
                swap   data[index – 3*step+1… index – 2*step] with
                       data[index – 2*step+1… index – step]
            }
            step = step*4;
        }
    }
}
```

Figure 9: Algorithm to put Data into Digit-Reversed Ordering given the Ordering produced by Architecture B Butterflies

In the recursive version of the algorithm we first looked at all the results and divided them into groups of N/4. Instead, the algorithm in Figure 9 processes each element starting with the first one produced by the FFT circuit using architecture B butterflies. Every 4 data points we swap points *data[index – 3]* and *data[index – 2]*. We then check if *index* is also a multiple of 16. If so we then swap blocks *data[index – 11…index – 8]* and *data[index – 7…index – 4]*. We continue to swap data in this manner as long as we detect that *index* is a multiple of a power of 4 that is less than num_points.

3.3.3 Addressing Design Considerations

We now address how butterfly architecture B affects the four parameters discussed in section 3.1. We begin with the *bit-width* parameter.

In this architecture the *bit-width* parameter has very little constraints. This is because only a single complex multiplier per FFT stage is used. This allows us to create complex multipliers with large precision without using too many device resources. While we do not have the luxury of using the shift-only-when-necessary scheme, we have the resources necessary to keep good quality of results by using more precision bits.

From the point of view of the number of points we can use, and thus the memory usage, we can use all memory available on the device. This is because the data input and output is serial, while the storage space is contained within the butterfly itself. No secondary buffers are necessary.

With all the benefits listed above, this architecture has one downside. The f_{op} requirement for this architecture has to match the sampling frequency, namely 200MHz. This is a problem as high-speed complex multipliers with large *bit-width* are very hard to create. This places a limitation on the bit-width parameter, requiring it to be sufficiently small to allow an implementation of a complex multiplier that runs at over 200MHz.

We will see in section 4 that it is indeed possible to design such a multiplier, but it is necessary to sacrifice some precision to do so. In anticipation of this design we now discuss how the complex exponentials, or Twiddle Factors, have to be generated for the butterfly architecture B. The design for the Twiddle Factor generation module will introduce additional constraints on device resources and parameters discussed in Section 3.1.

3.4 Twiddle Factor Generator

The Twiddle Factor generator used by architecture B butterfly is the CORDIC rotator architecture [5]. This architecture suffers from slow operation due to its iterative nature as well as computationally expensive hardware. To meet the timing requirements as well as the hardware resources limitations, an alternate Twiddle Factor generator architecture is proposed. The latter outperforms the CORDIC architecture in terms of speed and simplicity of design.

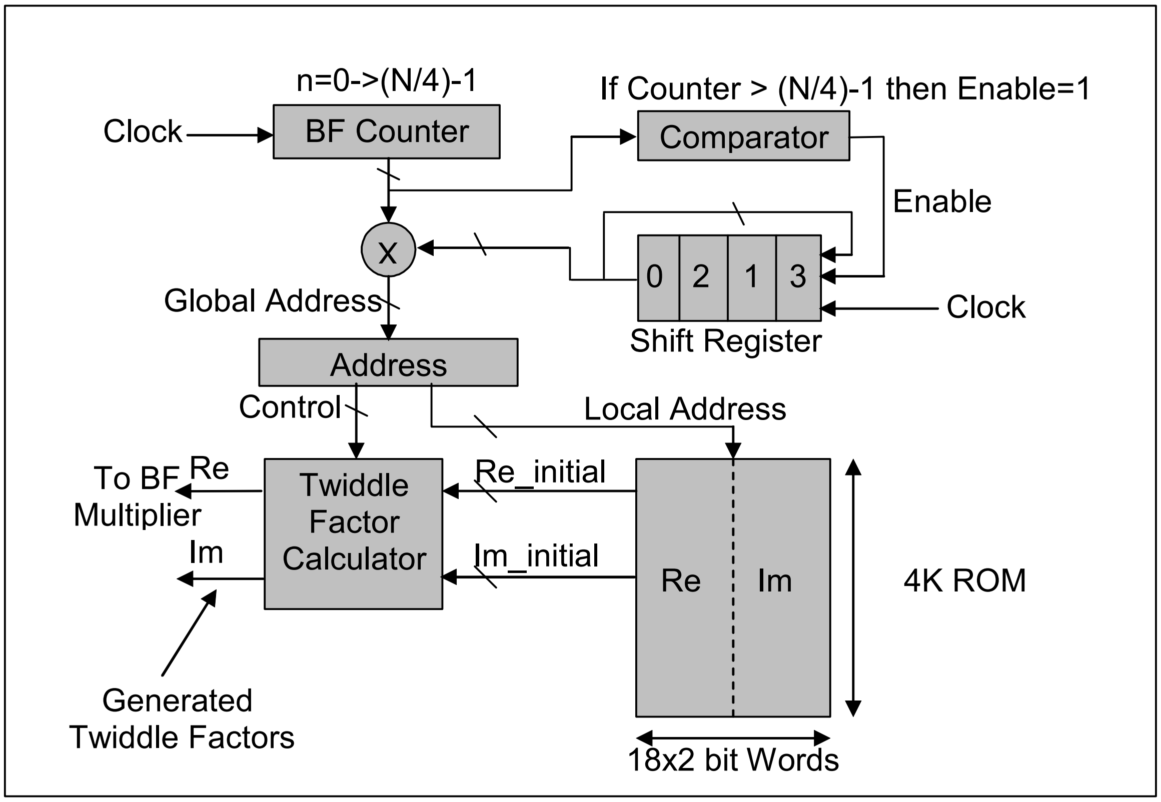

Figure 10: Proposed Twiddle Factor Generator

Stage	1	2	3	$(\log_2 N)/2$
Butterfly Groups	1	4	16	N/4
Butterflies per Group	N/4	N/16	N/64	1
Twiddle Factor Exponents				
leg 1	0	0	0	0
leg 2	n	4n	16n	(N/4)n
leg 3	2n	8n	32n	(N/2)n
leg 4	3n	12n	64n	(3N/4)n
	n=0 to N/4-1	n=0 to N/16-1	n=0 to N/32-1	n=0

Table 3: Butterfly Relations

We improve on the CORDIC rotator architecture by storing Twiddle Factors in Read-Only Memory (ROM). Figure 10 shows an abstract view of the first stage Twiddle Factor generator proposed in this work. The Twiddle Factor generator consists of a *butterfly counter*, a *shift register*, a *comparator*, a *multiplier*, an *address decoder*, a *Twiddle Factor calculator unit* and the *storage memory*. The butterfly counter is used to index of the element currently being processed by the butterfly. The shift register stores the coefficients corresponding to the group number (A0 through A3) being processed, as described in section 3.3.2. The comparator is used to inform the shifter when to shift. The shift of the coefficient happens when processing a complete bundle terminates. When both values are multiplied using the multiplier, the result will give the Twiddle Factor exponent showed in Table 3. The latter is used to fetch the corresponding Twiddle Factor in memory as described above.

The data flow involved with computing a Twiddle Factor is as follows. The counter (BF counter) iterates through the butterflies, starting at 0 and ending at the last index of the butterfly in a group, as per Table 3. This address is multiplied by a factor that corresponds to the leg currently being processed. For the first stage this means that the multiplication is by 0, 1, 2 or 3. The resulting address, referred to as global address, is the address in memory where the Twiddle Factor with the index corresponding to the global address is stored. We access the ROM to retrieve this Twiddle Factor and pass it to the butterfly complex multiplier unit.

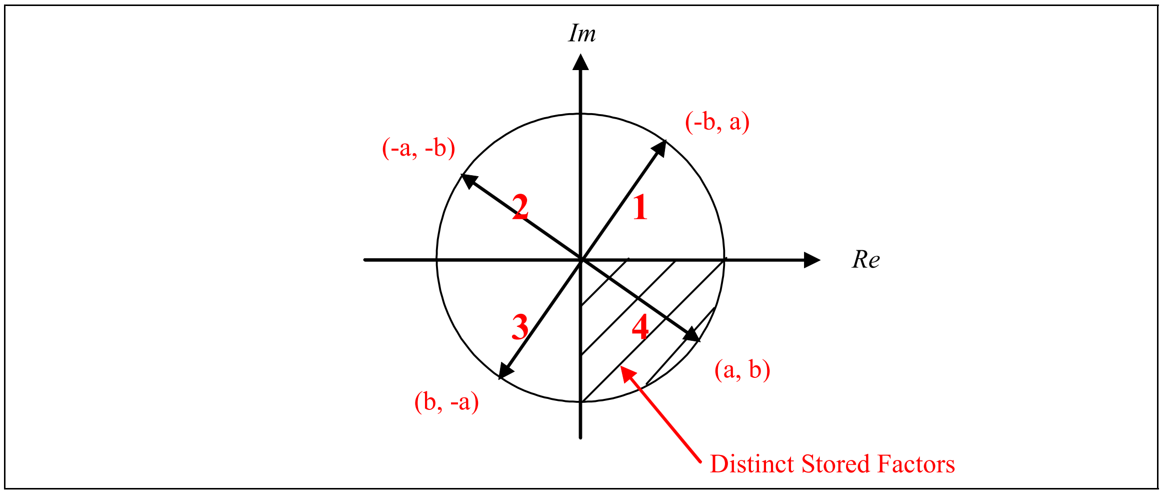

Figure 11: Twiddle Factor Calculation

To reduce memory usage we notice that only a quarter of the Twiddle Factors need to be stored as every other Twiddle Factor can be reconstructed having only the Twiddle Factors of the 4th quadrant. Depending on which quadrant other Twiddle Factors belong to, they can be calculated in terms of Twiddle Factors in any other quadrant of the unit circle. The computation of these Twiddle Factors is shown in Figure 11.

Thus we introduce the Twiddle Factor Calculator module that given values (a, b) and the quadrant where the corresponding Twiddle Factor is located, can calculate the value of the Twiddle Factor needed by the butterfly complex multiplier. It turns out that to obtain the values (a, b) and the quadrant for the final Twiddle Factor we only need to examine the global address. The two most significant bits of the global address specify the quadrant where the Twiddle Factor is located (00 for 4th quadrant, 01 for 3rd, 10 for 2nd and 11 for 1st) and the rest of the global address specifies the location of the (a, b) Twiddle Factor in the ROM. This is because we notice that the relationship between indices of Twiddle Factors (a, b) and (b, -a) is that if (a, b) has index *i* and there are a total of N Twiddle Factors then (b, -a) Twiddle Factor has index *(i+N/4)*.

We further notice that for each consecutive stage of the FFT, the number of Twiddle Factors required for computation decreases. This is because the Twiddle Factor Exponent index, as shown in Table 3, for stage i>1 is related to the Twiddle Factor Exponent index for stage i-1 by a factor of 4. Thus, for stage i, we only need to store every 4th exponent stored in stage i-1.

3.5 Matlab Analysis and Simulation

To better evaluate each of the radix-4 butterfly architectures we designed a Matlab model for them. Using these models we were able to produce results with a specific bit-width and N parameters. We compared these results to the floating point precisions results obtained by Matlab's fft() function to determine the inaccuracy introduced by using fixed-point computation versus floating point available in Matlab.

To model the butterfly architecture accurately we have developed Matlab code that operates exactly as we intended our circuit to work. We have therefore created binary adders, multipliers and registers to be able to observe how the variation in bit-width and N parameters affected the final results. We used these models to perform simulations and obtain results that we could use to determine the proper values for bit-width and N parameters. These results are summarized in the following subsections.

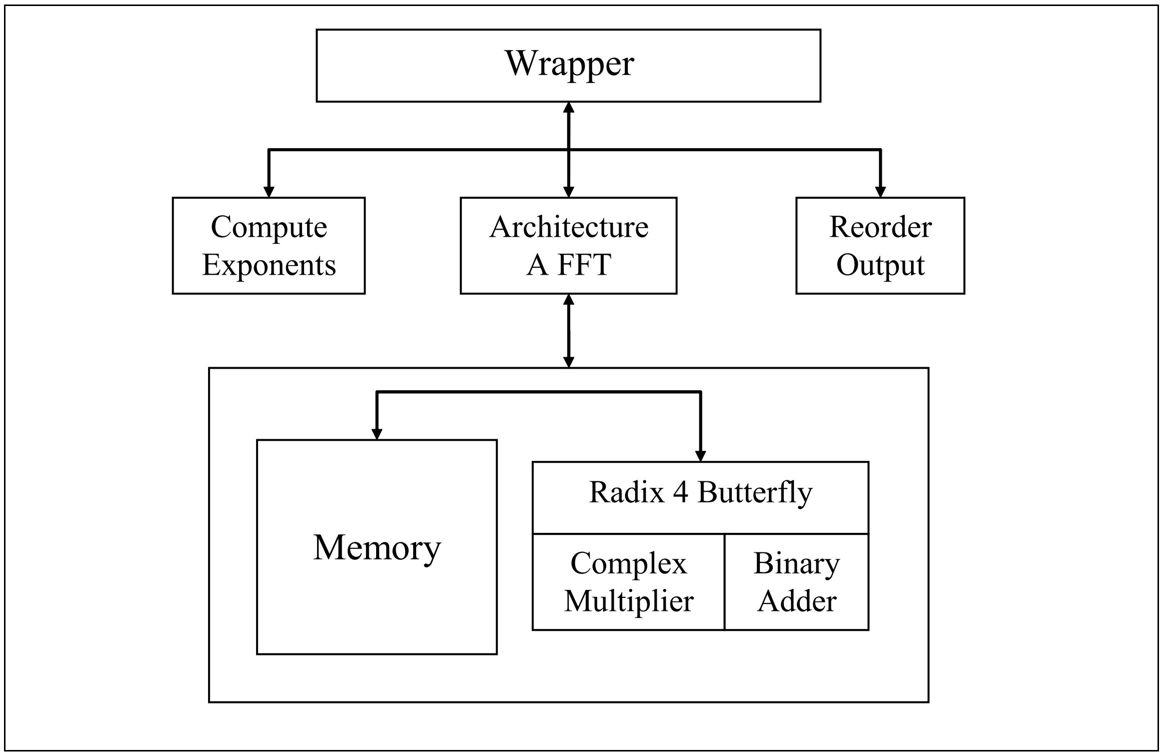

Figure 12: Model for Architecture A FFT

3.5.1 Butterfly Architecture A model simulations

The architecture A model is depicted in Figure 12. At the top level we have a wrapper that allows the model to be used to compute the FFT. In the wrapper we change the floating-point data into fixed-point data and create the Twiddle Factors. This data is then passed into the FFT block for processing.

Inside the FFT block we created local memory for data and Twiddle Factor storage. To compute the FFT we instantiate a radix-4, architecture A, butterfly and iteratively process 4 data points at a time, reusing the memory buffer to limit the memory requirement. The butterfly itself uses several adders and multipliers to compute the intermediate values, as shown in Figure 12.

Finally the results of the FFT processing are returned to the wrapper. In the wrapper the output data is converted back into floating point format and reordered to obtain meaningful data representation.

We ran several tests on this butterfly architecture and compared the results we obtained to the ones generated by Matlab. As shown in Figure 13 and Figure 14, the results for 9-bit data precision and for 256 and 1024 point FFT are promising. However, at 1024 points the distortion is noticeable. This means that for 16384 point FFT more data precision needs to be used.

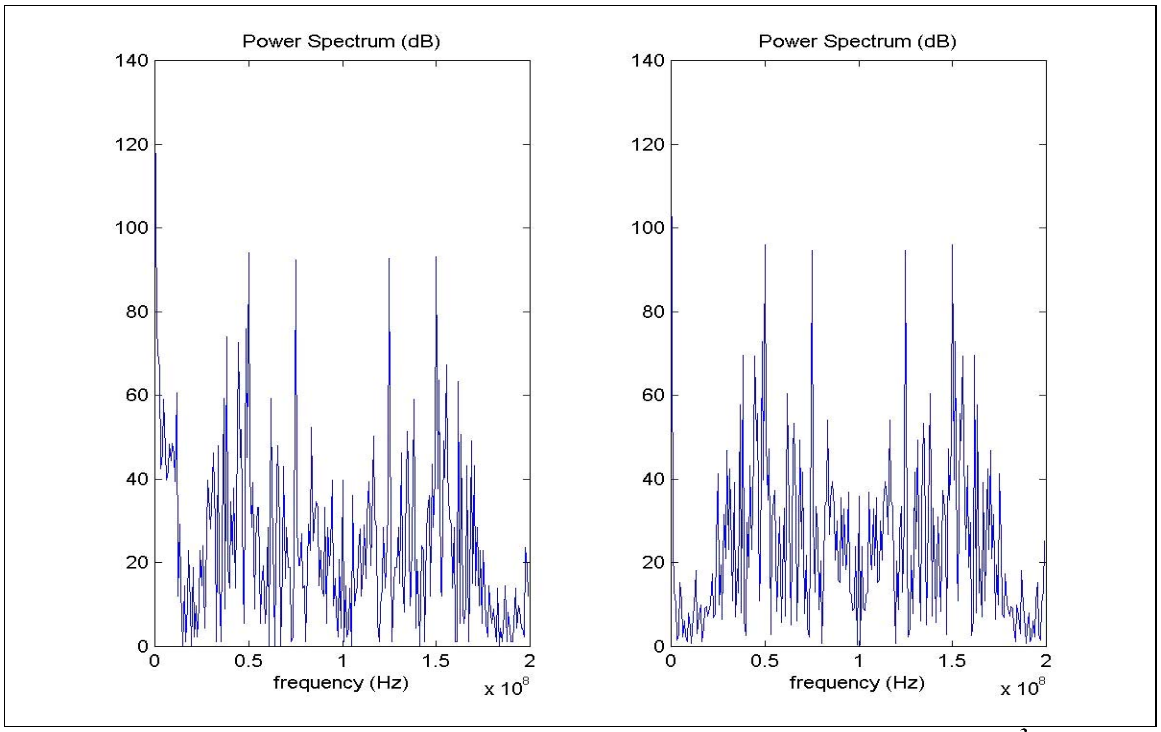

Figure 13: Results using architecture - A FFT for 256 points at 9-bit precision[3]

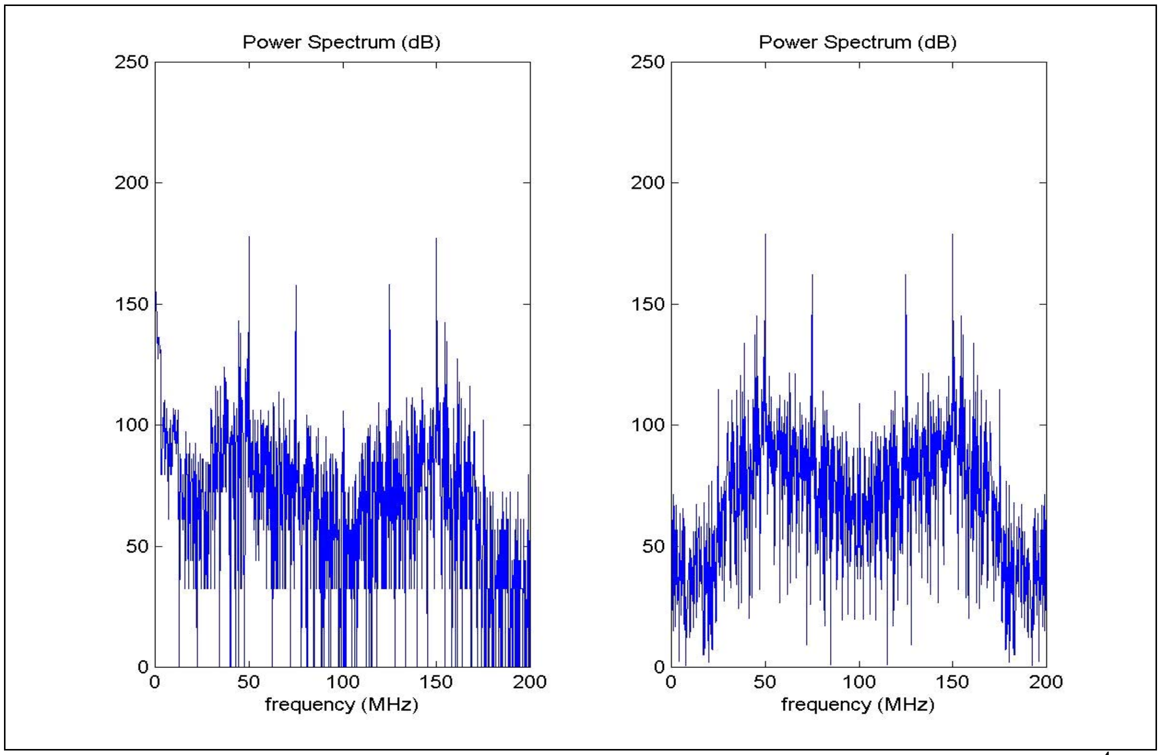

Figure 14: Results of simulations for 1024 points at 9-bit precision using architecture A FFT[4]

---

[3] The left image is produced by our model, while the right image is that produced by Matlab

[4] The results on the left are produced by out model, while the results on the right are produced by Matlab

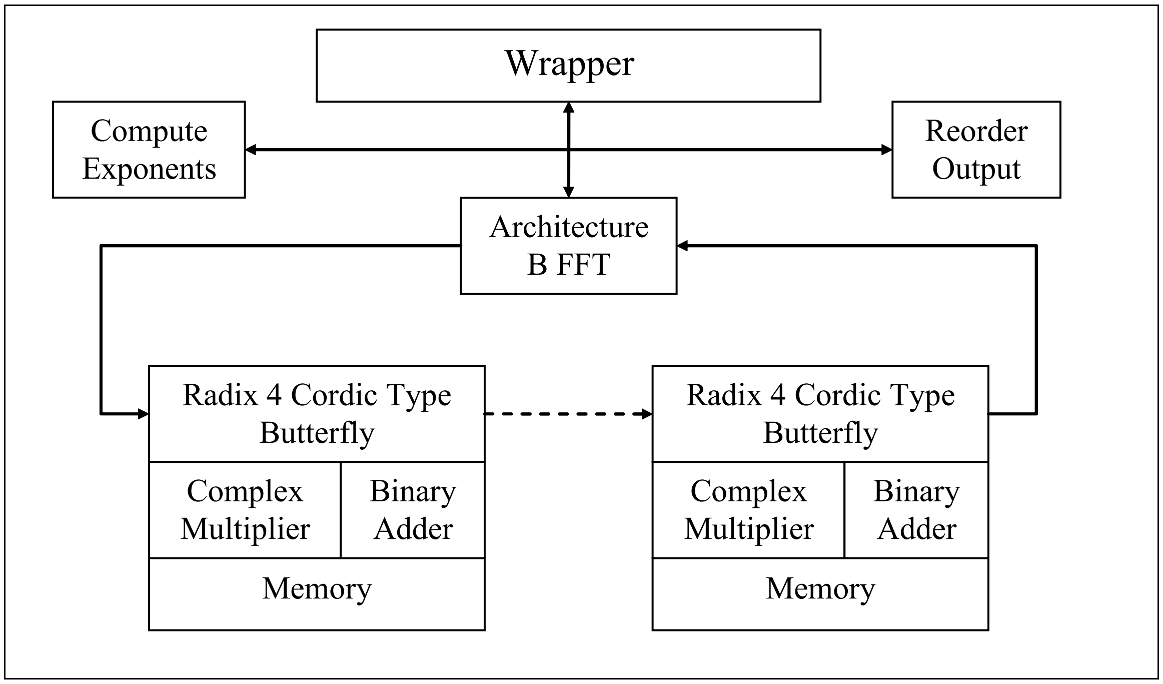

Figure 15: Architecture B FFT Model

This observation presents a problem, as increasing the bit-width parameter past 9 will double the number of DSP multipliers used by the chip. With 12 DSP multipliers used per butterfly at 9-bit precision and a limit of 80 DSP multipliers on the Stratix 20K device, we could only fit 3 butterflies on the device, with precision of up to 18 bits. This presents a throughput limitation, which will be addressed during preliminary hardware design.

3.5.2 Butterfly Architecture B model simulations

The architecture B model is depicted in Figure 15. The top level wrapper is the same as for architecture A, but the organization of components is different. In the architecture A the key was to use several butterflies in parallel to compute portion of the FFT at a single stage. In the architecture B we devote a single butterfly to compute all results for a single stage.

In this architecture the data is passed from one stage to the next, allowing the FFT to be computed one data point at a time in a large pipeline. The output of the FFT starts appearing after all data points have been entered into the FFT. Thus, the FFT itself looks like it is pipelining complete FFT computations, as an output of one FFT is being produced as the second FFT starts being processed.

While for this architecture we do not have the shift-only-when-necessary and we need to shift data two bits to the left at the output of each stage. While this introduces errors, for 18-bit data precision and 16384 points the results have good quality. As shown in Figure 16 the noise introduced to the computation is mostly below -10dB.

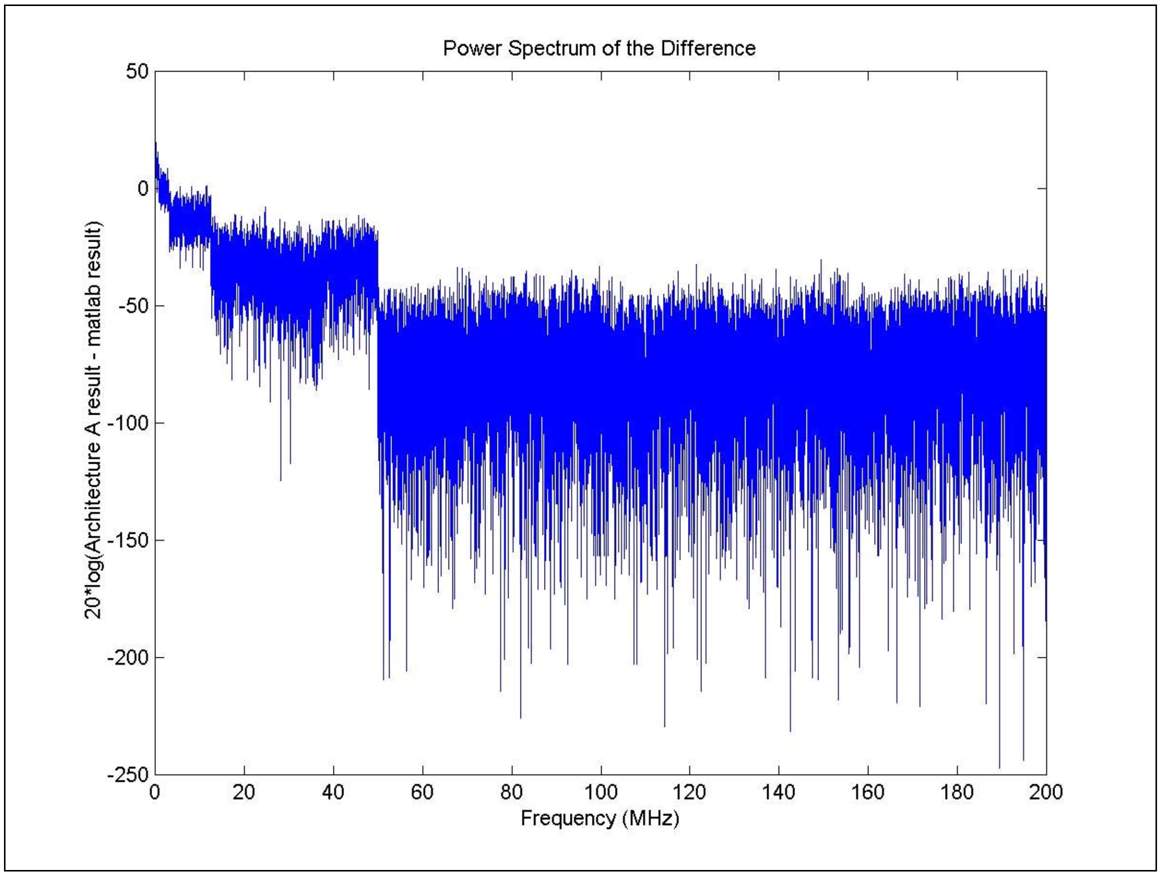

Figure 16: Power Spectrum difference between Matlab computed FFT and FFT computed by architecture B

4. Hardware Design

In this section of the report we discuss the hardware implementations of the two FFT architectures we discussed in section 3. The first implementation, architecture A, uses a more common version of a radix-4 butterfly. We augment architecture A with a feature that improves the quality of results for signals with flat frequency spectrum. The second architecture, referred to as architecture B, is based on the paper by Despain [5] and uses only a single multiplier per radix-4 butterfly. Through the analysis of results obtained after implementing these architectures in VHDL Hardware Description Language we select the architecture that is best suited to implement the FFT algorithm using available resources.

4.1 Architecture A

The transition from the Matlab code to the Hardware Description Language definition of a circuit is relatively straightforward. The key here is to specify the location of registers and setting compiler directives to preserve their placement. In the architecture A the implementation is not particularly complex. The register distribution is shown Figure 17.

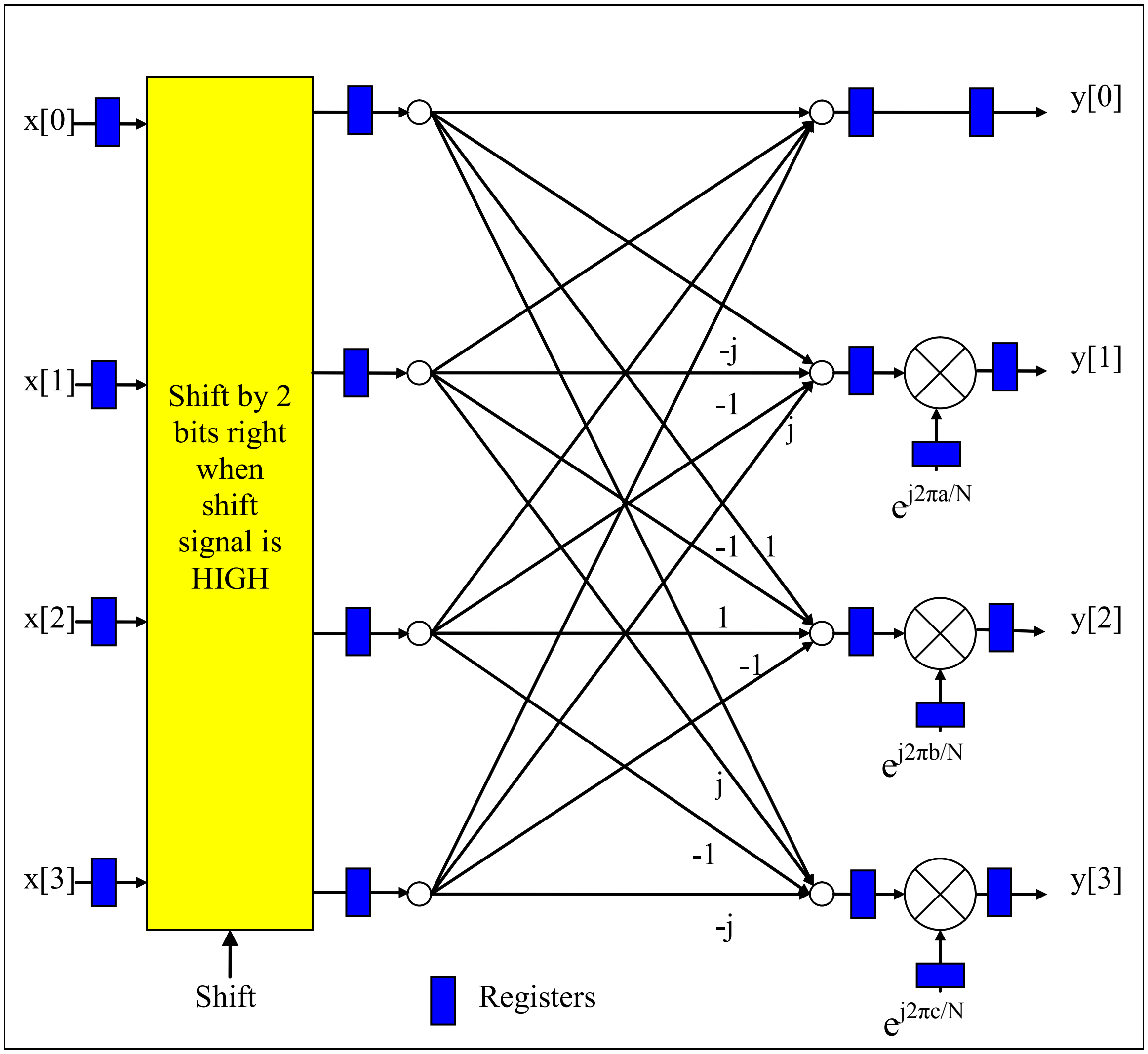

Figure 17: Pipelined Architecture A Butterfly Implemented in Hardware

The initial implementation of this architecture did not include any compiler directives, which unfortunately caused the circuit to work barely at 75MHz. This speed was not acceptable; hence two improvements to the circuit design were made. The first improvement was to enforce the separation of the complex multiplier unit into two sub units – one to compute the real part and the other to compute the imaginary part of complex number multiplication. The second modification was to reduce the precision of the adder/subtractor used within the complex multiplier.

The first modification was necessary to force the Quartus II CAD tool to separate the real and imaginary computations. This in turn placed these respective modules in separate DSP multiplier blocks improving the overall performance of the circuit but resulting in extra hardware overhead. The second modification comes from the realization that a 36-bit adder is slow. To improve its speed we notice that after the complex multiplication the data being stored is only 18-bits. Thus, the least significant bits in the operation are removed. Since they are removed from further processing, it is not necessary to use all 36 bits for addition/subtraction. The error introduced in this adder shortening is negligible.

Registers were inserted in the design as indicated in Figure 17, to decrease the critical path and hence, increase the operating frequency of the design. As a result, architecture A butterfly was found to operate at 153.96MHz at 9-bit precision and 91.32MHz at 18-bit precision. Our results in section 3 for the architecture A showed that more than 9-bits would be required to obtain reliable results, with most desirable results at 18-bit precision. However, at 91.32MHz initial maximum operating frequency, it is more likely that the design will deteriorate in speed as the FFT includes more and more stages. On a larger FPGA where we could fit more Architecture A butterflies, this design would be feasible. However, the Stratix 20k has limited resources. Thus, because of insufficient hardware resources requirements, we turn our hopes to architecture B.

4.2 Architecture B

To implement the architecture B in hardware we followed the same process as for architecture A. In addition to pipelining the design, some of the registers previously allocated to the shift register blocks needed to be moved to act as input and output registers to smaller shift-register blocks. This modification was necessary to preserve the operation of the butterfly, while increasing the operating frequency of the design. The modifications to the design and location of the inserted registers are shown in Figure 18.

The resulting operating speed for this butterfly architecture is 248MHz. While we did need to use the same adder modification as for architecture A, this architecture managed to meet timing and resource requirements. With this module built, we now turn our attention to the creation of a Twiddle Factor generator module that can match the speed performance with this butterfly design to complete the FFT project.

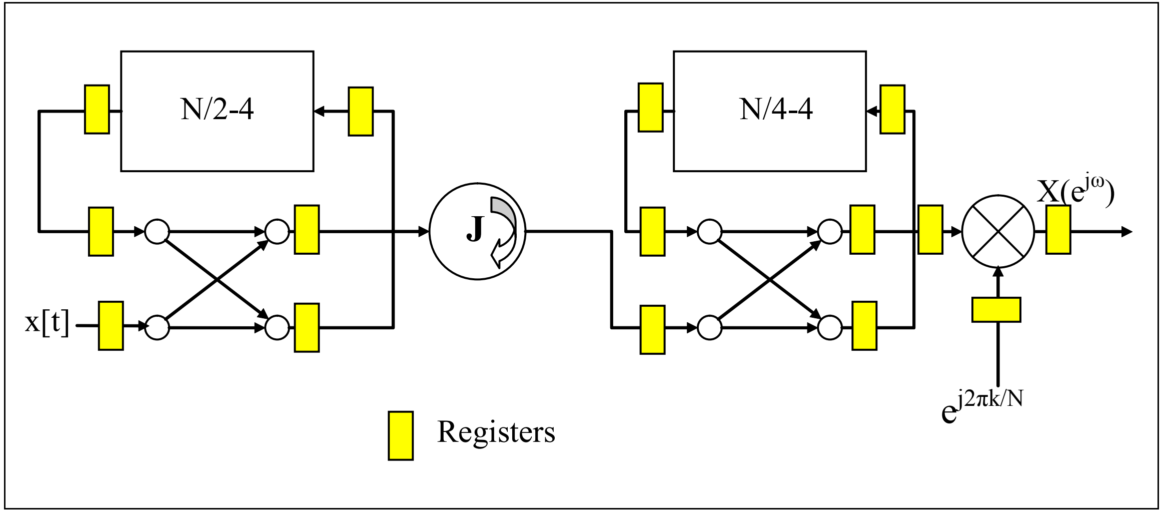

Figure 18: Architecture B butterfly as implemented in hardware

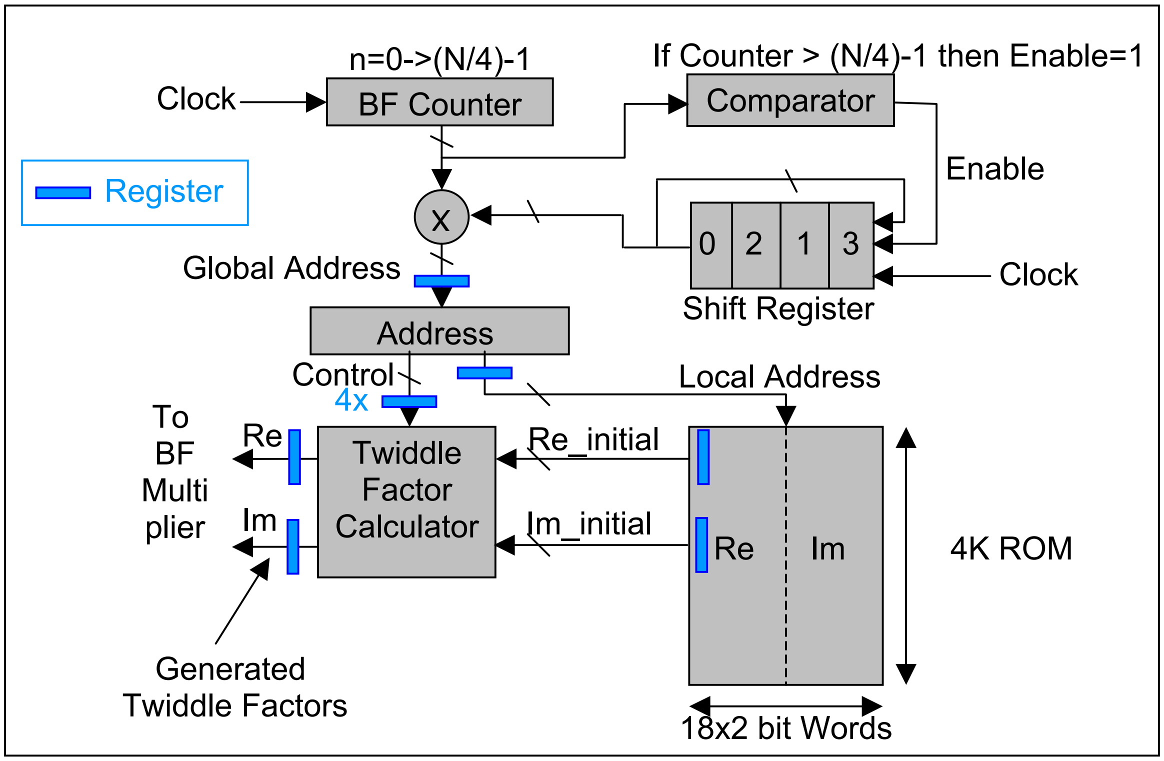

Figure 19: Pipelined Twiddle Factor Generator for Stage 1

4.4 Twiddle Factor Generation for Architecture B

The actual implementation of the Twiddle Factor generator is described in this section with emphasis on timing and hardware resources issues. The Twiddle Factor generator proposed in this work is attractive since it meets the timing requirements and offers considerable hardware savings especially in terms of storage. In general, N/2 Twiddle Factors are usually stored and the rest is computed. However, the proposed architecture needs only to store N/4 factors. For the following stages, the required number of factors to be stored shrinks considerably if the shift register is programmed with the same coefficients instead of the coefficients shown in Table 3. This is because the use of the coefficients shown in this figure, for later stages, results in spaced generated addresses, which end up with wasted memory locations that will not be accessed anyways. In order to meet the timing requirements, the Twiddle Factor generator is pipelined. The pipelined version of the design is shown in Figure 19.

The size of the BF counter is equivalent to M, where 2^M is the number of butterflies per group, and when multiplied by 2-bit shifter coefficient results in M+2 bit Global address width. This M+2 bit Global address is broken into M-bit Local address and 2-bit control code, as shown in this Figure. As described in Section 3.4, the former is used to access on e of the stored N/4 distinct Twiddle Factors while the latter is used to compute the actual factor to feed the Butterfly based on the quadrant it belongs to. The complex Twiddle Factor consists of two 18-bit real and imaginary parts, which are stored in a 36-bit wide ROM.

4.5 Register Access

As previously discussed, each Twiddle Factor generator and butterfly module has several internal registers. The primary purpose of these registers is to pipeline the computation, and hence reduce the operating frequency. However, such registers have a secondary purpose, as they may be used for testing purposes as internal access points within the design. Having read/write access to such registers results in an increased *observability* and *accessibility* to the design. If there is a problem internal to the design, the tester could write data to these registers, allowing the design to process the written data, then read results from other internal registers further down the pipeline. As a result, the current pipeline registers are given read/write access from the top level. As well, several other registers were added to the system to be used as internal access points.

Figure 20 and Figure 21 show the register access points of the Twiddle Factor Generator and the Butterfly Module, respectively. Furthermore, Table 4 and Table 5 describe the registers at each address for these two modules.

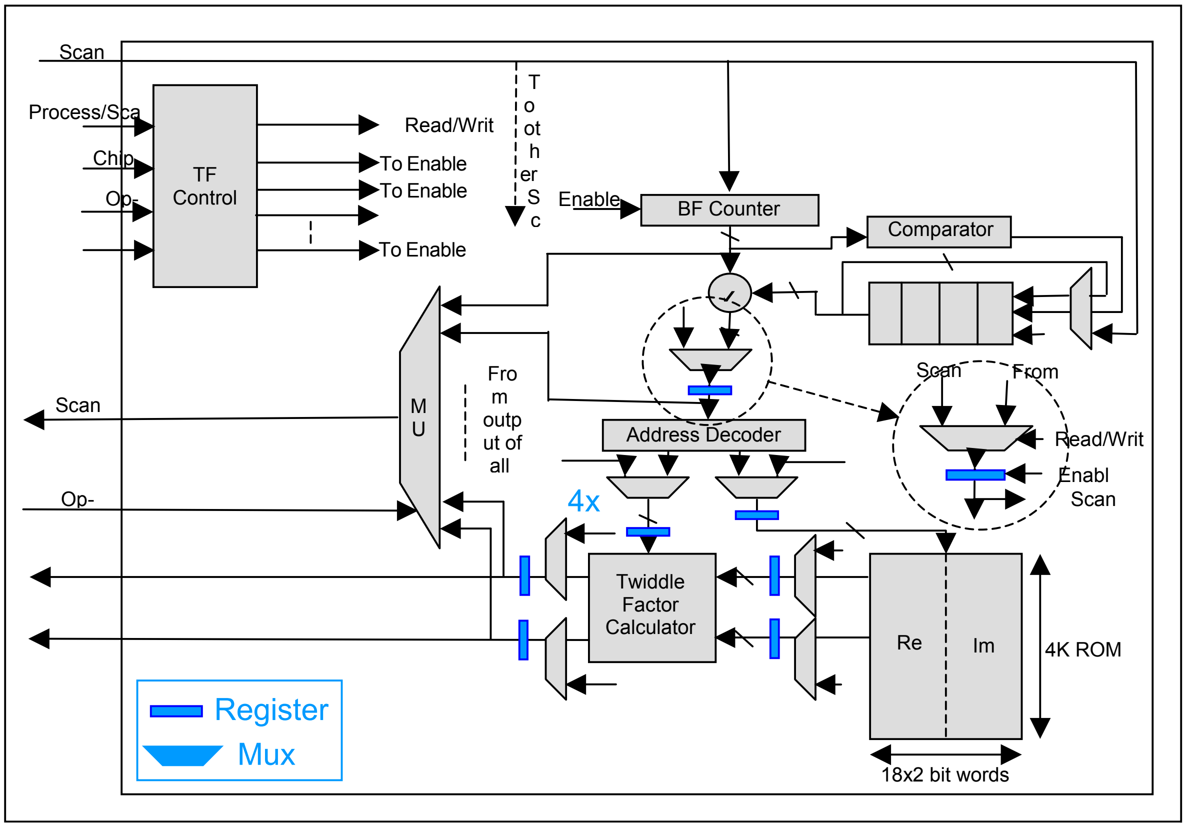

Figure 20: Register Access Points of Twiddle Factor Generator

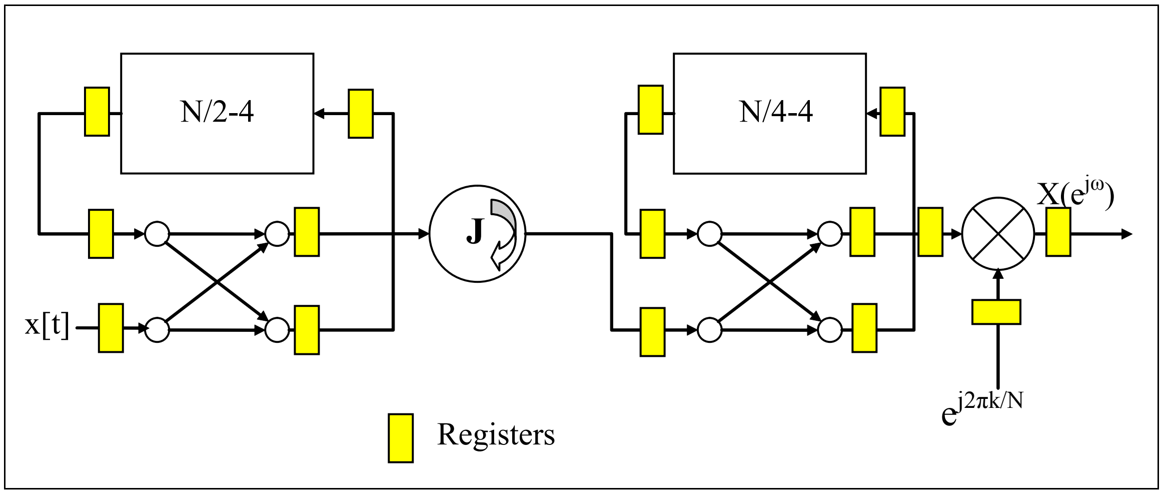

Figure 21: Register Access Points of Butterfly Module

Address	**Description**
0x0	Butterfly counter register
0x1	Shift register
0x2	Global address register
0x3	Local (ROM) address register
0x4	Ctrl register #1 [5]
0x5	Ctrl register #2
0x6	Ctrl register #3
0x7	Ctrl register #4
0x8	Real-part ROM output register
0x9	Imaginary-part ROM output register
0xa	Real-part Twiddle Factor register
0xb	Imaginary-part Twiddle Factor register

Table 4: Register Access Points of Twiddle Factor Generator

Address	**Description**
0x0	Input to butterfly (real)
0x1	Input from big memory (real)
0x2	Input from small memory (real)
0x3	Twiddle Factor inputs (real)
0x4	Output to big memory (real)
0x5	Output to small memory (real)
0x6	Input from j multiplier (real)
0x7	Final butterfly output (real)
0x8	Input to butterfly (imaginary)
0x9	Input from big memory (imaginary)
0xa	Input from small memory (imaginary)
0xb	Twiddle Factor inputs (imaginary)
0xc	Output to big memory (imaginary)
0xd	Output to small memory (imaginary)
0xe	Input from j multiplier (imaginary)
0xf	Final butterfly output (imaginary)

Table 5: Register Access Points of Butterfly Module

[5] The # refers to the stage or the register number of the pipelined Ctrl signal.

Stage	**Component**	**Address Space**	
		Start (hex)	**End (hex)**
1	Twiddle Factor	0x00	0x0f
	Butterfly	0x80	0x8f
2	Twiddle Factor	0x10	0x1f
	Butterfly	0x90	0x9f
3	Twiddle Factor	0x20	0x2f
	Butterfly	0xa0	0xaf
4	Twiddle Factor	0x30	0x3f
	Butterfly	0xb0	0xbf
5	Twiddle Factor	0x40	0x4f
	Butterfly	0xc0	0xcf
6	Twiddle Factor	0x50	0x5f
	Butterfly	0xd0	0xdf
7	Twiddle Factor	0x60	0x6f
	Butterfly	0xe0	0xef

Table 6: System-Level Multiplexing of Register Access Points

As previously discussed, each Twiddle Factor and butterfly module has several internal registers, which are read-write accessible at the top-level. To access all registers in each module, four bits of address space are required. Recall, to implement a 16k point FFT, seven stages is required. Hence, seven Twiddle Factor modules and seven butterfly modules are present in the system. For a total of 14 modules, another four bits of address space are required. With this in mind, we may derive a system-level register map that distinguishes the registers of each module. Table 6 summarizes this register map. Note that the address ranges 0x70-0x7f and 0xf0-0xff are currently unoccupied. Some of these addresses are discussed in later sections. Others are left blank for future use.

5.0 System Integration

We will now describe how the FFT system described above is incorporated into the FPGA. We will first describe how the FFT is encapsulated to sit on a global bus running around the system, called *the Avalon bus* [7]. We will then describe how this new system, called the Avalon bus model, is used in a complete system-on-chip environment using Altera's tool *SOPC Builder* [3]. We will finally describe how this system, called the embedded system, is interfaced to the host computer for real-time verification.

5.1 Integrating the FFT System

Early in the design stage, it was determined that the best way to perform real-time hardware testing was to instantiate the FFT system on an Avalon bus. Using Altera SOPC Builder, a NIOS processor could master this bus, and could perform Avalon reads and writes to the FFT. Two types of data could be read or written to the system.

(1) Register access data
(2) FFT incoming and outgoing data

To perform reads and writes of these data, a wrapper surrounded the FFT, which implemented the following functionality.

(1) Avalon reads and writes do not exactly translate into a register read or write. Therefore, some basic logic was necessary to translate an Avalon read or write into one that would correctly read or write to an internal register.

(2) The bus operates at either a 25 or 50 MHz clock rate. However, the FFT has harder timing constraints since it has to operate at a frequency four times higher than the bus speed. Hence, asynchronous clock-domain logic was necessary for some select non-combinational signals.

(3) To handle the real-time processing of incoming and outgoing data in the FFT, incoming and outgoing data had to be buffered between the NIOS processor and the FFT. Furthermore, additional addresses were added to the above address map to handle real-time processing. This additional functionality is described in further detail below.

Recall, the NIOS processor sends data to the FFT for data processing. At first glance, one might argue that the NIOS could stream data to the FFT for processing. However, the following two factors impede the real-time processing performance of the FFT instantiated on the Avalon bus.

(1) The FFT operates at a clock rate approximately 4 times faster than the bus clock rate.

(2) Because the NIOS processor must decode instructions, the NIOS is not capable of sending data along the bus every clock cycle.

Because of these two factors, the following logic was implemented to perform real-time data processing on the FFT.

(1) The NIOS processor would first send data to an input FIFO. This input FIFO would buffer data prior to it being passed into the FFT.

(2) Once sufficient input data has been buffered, the NIOS processor would write an unsigned 13-bit count to a predetermined address. This would initiate a "Go" instruction, which would enable the FFT computation (or processing if you wish) and pass the incoming data into the FFT at a rate of one sample per clock cycle, until the count has expired. While data is being passed through the FFT, outgoing data would again be buffered in an outgoing data buffer, at a rate of one sample per clock cycle.

(3) Following completion of the "Go" instruction, outgoing data would be collected from the outgoing buffer by the NIOS processor, from where it could be analyzed further.

Based upon the above logic, several more registers were added to the register map, as can be shown in Table 7.

Address	Description
0xf1	Writing to this register adds a sample to the tail of the real input buffer
0xf3	Writing to this register adds a sample to the tail of the imaginary input buffer
0xf2	Reading from this register collects a sample from the head of the real output buffer
0xf6	Reading from this register collects a sample from the head of the imaginary output buffer
0xf4	Writing to this register initiates a "Go" instruction, where the 13-bit unsigned value written tells the system how long to execute the "Go" instruction for. Because the input and output FIFOs are only 256 words in depth, the maximum length of a "Go" instruction is 256 cycles.

Table 7: Registers for "Go" Instruction Implementation

Figure 22 shows how the additional "Go" logic wraps around the existing FFT architecture. In summary, if a write is issued to either of the input FIFOs, then the input FIFO control block would assert the respective input FIFO enable signal high. The incoming FIFO data would then be added to the tail of the respective FIFO. Each input and output FIFO is only 256 words in depth. Hence, this is the upper bound to how many words may be added to each FIFO.

Once the input FIFOs have been filled, writing a count to address 0xf4 initiates a "Go" instruction. The "Go" control block receives the incoming data from the NIOS, which would be a 13-bit number representing the number of clock cycles to enable the FFT. The "Go" control block would then enable the input FIFOs, the FFT and the output

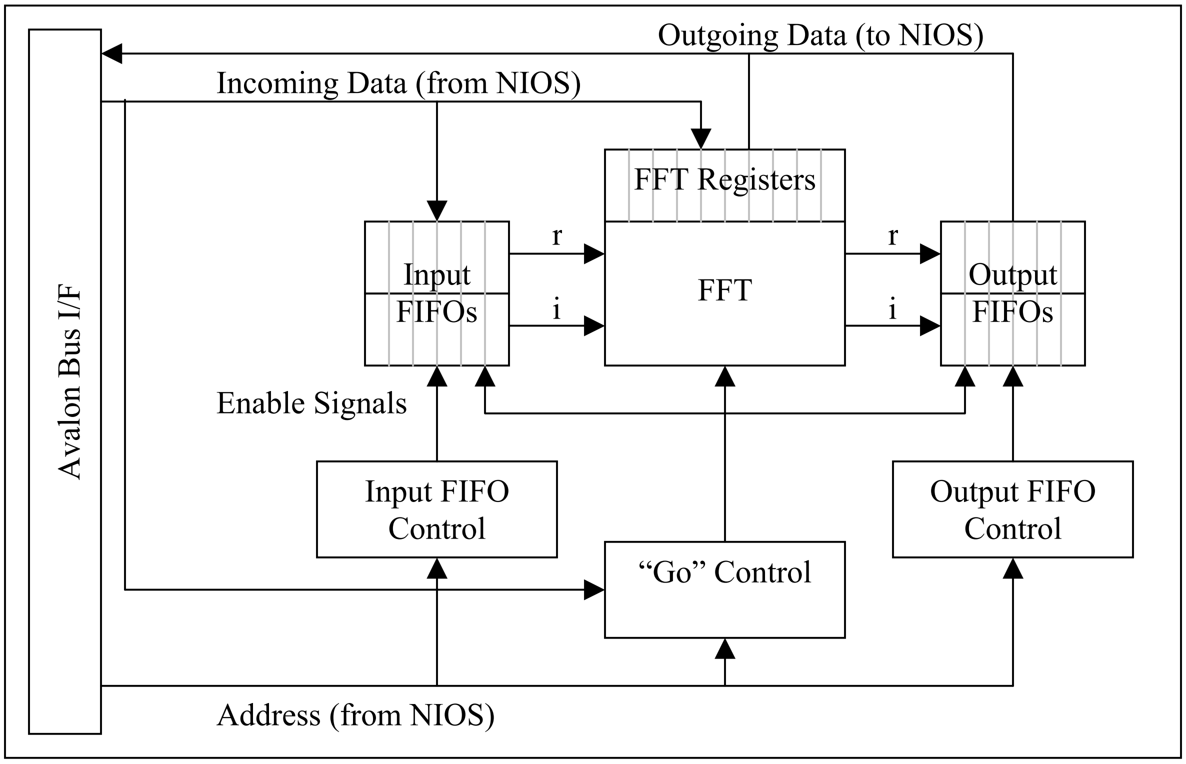

Figure 22: "Go" Logic Implementation

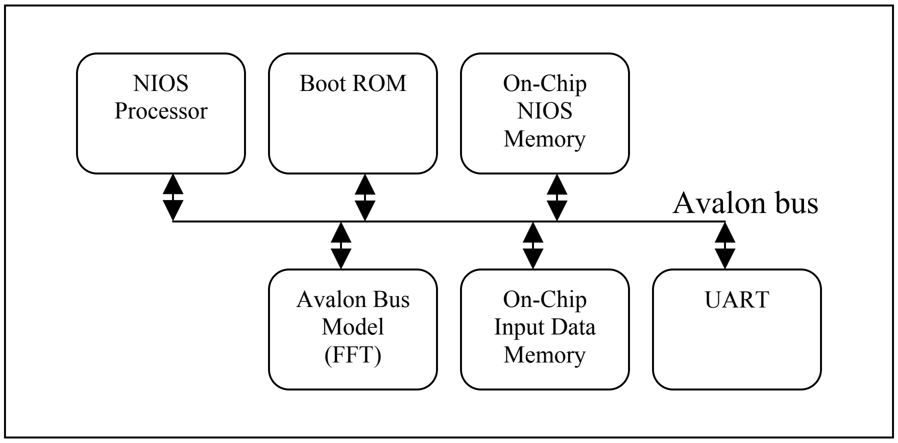

Figure 23: System-Level Implementation

FIFOs. The input data would be passed from the input FIFOs to the FFT. The FFT would process the data, and produce output data, which would be stored in the output FIFOs.

Finally, if a read is issued to either of the output FIFOs, then the output FIFO control block would assert the respective output FIFO enable signal high. An output sample would be collected from the head of the respective output FIFO. From this point onward, this system will be referred to as the Avalon bus model.

5.2 Integrating the Avalon Bus Model

Altera SOPC Builder is a tool that is used for integrating several cores along a common bus, such as an Avalon bus. The following diagram shows how this tool was used to integrate the Avalon bus model to a NIOS processor, memories, and a UART. Each of these components may be described in further detail.

NIOS Processor (0x2100-0x21ff): The NIOS processor is the only master on the Avalon bus. The NIOS processes software code, initiating data reads and writes to other bus cores as necessary.
Boot Rom (0x0000-0xffff): Upon initial system reset, the NIOS processor begins fetching instructions from the boot ROM. The boot ROM will contain one of two types of software code:

(1) An application-specific software code, which the NIOS will immediately begin executing.
(2) The Altera GERMS monitor. The GERMS monitor is primarily used for debugging purposes. Via the UART or the JTAG interface, you can use the GERMS monitor to view all memory locations on the bus. Using the GERMS monitor, you can also load, debug and step through software code in a fine-level of detail.

On-Chip NIOS Memory (0x1000-0x1ffff): The NIOS processor requires another writeable memory for its vector table, program memory and data memory.

Avalon Bus Model (FFT) (0x3000-0x3fff): This is the location of the Avalon bus model, which includes the FFT and all surrounding Avalon bus wrappers.
On-Chip Input Data Memory (0x4000-0x7fff): The basic three steps of the NIOS processor are as follows:

(1) Write 256 input data samples to the FFT
(2) Send a "Go" instruction
(3) Read 256 output data samples from the FFT

To send data from the FFT, there are several possible methods (which will be discussed later). However, the best and final method was to pre-load input data into a dedicated memory. This preloaded data would be retrieved from the NIOS, then transmitted to the FFT. This on-chip input data memory contains 16384 8-bit real input values.
UART (0x2000-0x201f): The UART is one of two possible communication means between the NIOS processor and the host computer. The other is the JTAG interface. Although the UART is not used in the final demonstration, it is used in Embedded System Simulation (to be discussed later).

Noticeably missing from the above diagram is any reference to clocking. Recall that in the above diagram, the bus clock is approximately four times slower than the FFT clock. Hence outside of the above system, a PLL is necessary.

The above system, combined with the PLL, is the complete system to be implemented on the FPGA. This system will be referred to as the embedded system.

5.3 Integrating the Embedded System

We can conclude by discussing how the embedded system is integrated with the host computer. Essentially, there are two possible paths of communication between the host computer and the embedded system:

(1) The UART interface
(2) The OCI interface, which connects to the NIOS processor via the JTAG interface.

Although the UART interface is much faster, the OCI JTAG interface was used for communication. This is because the OCI JTAG interface is primarily intended for debugging purposes. Hence, there are more debugging scripts which came in handy. An example of this is the "stdio window" script, which launches a separate standard I/O window. This window can be used for sending data from a file, or logging data to a file. Hence, for hardware verification, and for the final demonstration, all communication is done with the OCI JTAG interface. At the host computer, connecting to the OCI interface is straightforward. Altera Perl scripts can be used to launch a command window as a path for communication.

5.4 Difficulties with Integration

Although it should be theoretically very simple, several unexpected problems were encountered at various points of system integration. The following points highlight several bugs that were fixed during the integration phase.

Bug #1:	The LPM RAM memory was not being instantiated by the tool correctly, resulting in correct Quartus simulator outputs, but incorrect Testbuilder simulator and Hardware outputs.
Fix #1:	Replace LPM RAM with AltSyncRAM. Both Tomasz Czajkowski and Chris Comis fixed this bug.
Bug #2:	There is a two clock cycle latency from when an address is passed to a Twiddle Factor module to when the data is outputted from the associated ROM. This latency was not taken into account for when the Twiddle Factor module was disabled. Hence, the Twiddle Factor module did not work when the enable was triggered off and on.
Fix #2:	Register the data from each Twiddle Factor module when the system is disabled. When the system is re-enabled, feed the registered data back into the pipeline.
Bug #3:	For the fifth butterfly stage, a small coding error resulted in a line being executed even if the circuit was disabled.
Fix #3:	The code was easily fixed.

6.0 Verification Methodology

Now that we have described the entire system, we will now describe the verification strategy used to verify the system. We will first briefly describe how the Quartus simulator was used for basic sanity checks. We will then describe two Testbuilder systems, which were used to verify the FFT system and the Avalon bus model, respectively. We will then describe the system-level simulation environment that was used to verify the embedded system. We will conclude by discussing how the entire system was verified in hardware.

6.1 Verifying Modules Inside the FFT System

For basic verification of the Twiddle Factor Generators and the Butterfly Modules, the Quartus simulator was used. This waveform simulator is integrated into the Altera Quartus environment, and can be quickly used to verify the basic functionality of small to mid-sized blocks. Figure 24 shows a screen capture of the Quartus Simulator.

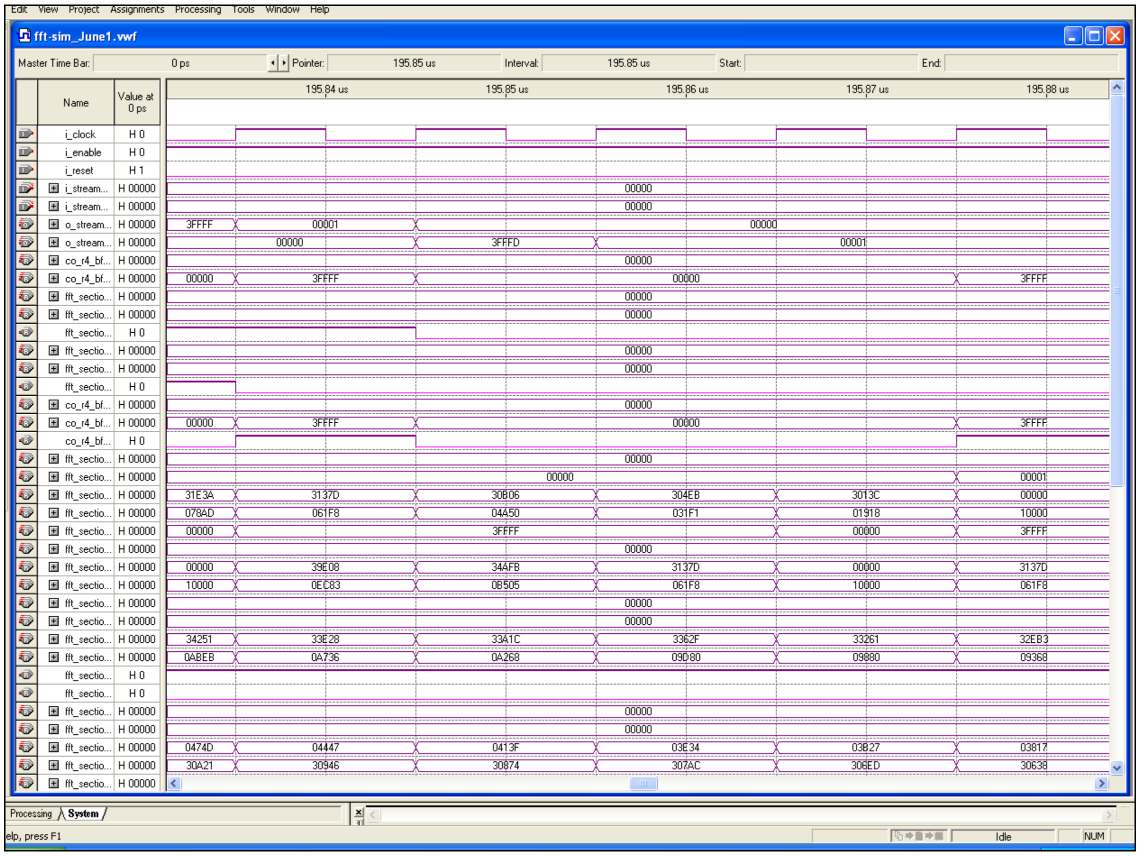

Figure 24: Screen Capture of Quartus Simulator

6.2 Verification of the FFT System

Although Quartus simulations can be used for basic sanity checks, a more detailed simulation environment is required for more extensive verification. Hence, Modelsim and Testbuilder were used for in-depth verification at the system-level. Modelsim offers several important advantages over Quartus simulation:

- In Quartus simulation, all flip-flops are by default reset to zero. Furthermore, a signal can either be only logic high (1) or a logic low (0).
- In Modelsim, all flip-flops are by default not reset to zero. As well a signal can be one of nine possible different values:
  - logic high (1)
  - logic low (0)
  - uninitialized (U)
  - unknown (X)
  - high impedance (Z)
  - weak unknown (W)
  - weak low (L)
  - weak high (H)
  - don't care (-)

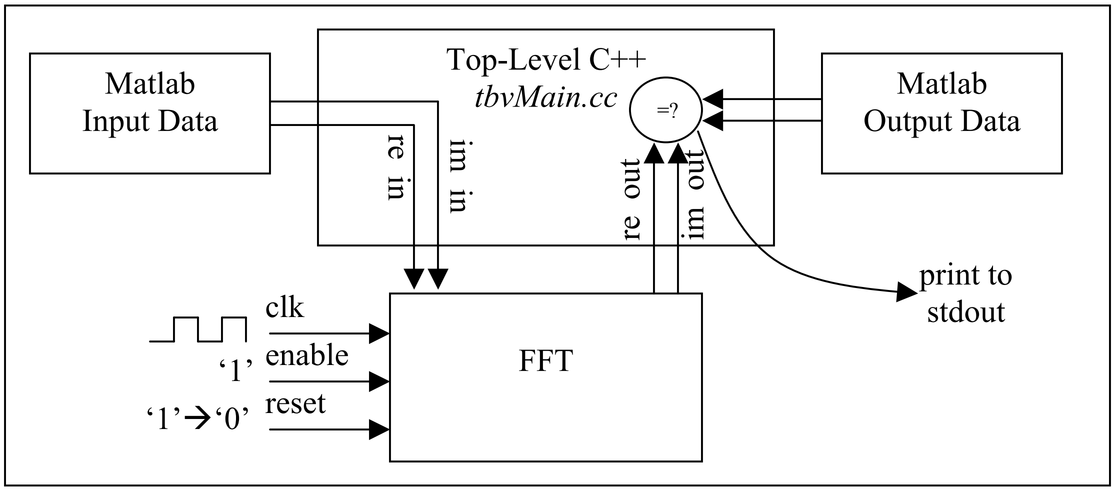

Figure 25: Simple Testbuilder Environment

For these reasons, Modelsim is arguably a more precise simulator.

- Modelsim simulation takes 15 minutes where an equivalent Quartus simulation would take over 2 hours.
- Quartus simulation was found to model some Megacore blocks incorrectly, where Modelsim, in conjunction with Altera libraries, would not.

Testbuilder is a set of C++ libraries that is used in conjunction with Modelsim, and allows verification to be done at the transaction level using C++. Testbuilder provides a more comprehensive environment for high-level self-checking verification.

For the first Testbuilder environment, the FFT system was instantiated. Figure 25 illustrates this first Testbuilder environment.

As can be shown, the first Testbuilder environment isolates only the FFT. This environment proved very useful for the following reasons:

- Matlab input and output data was easily imported into the testbench
- The self-checking mechanism was easily coded at the C++ level
- Modifications were easily written at the C++ level. For example, the system could be frequently disabled and re-enabled to check the enable features of the FFT.

6.3 Verification of the Avalon Bus Model

The second Testbuilder environment was much more elaborate. The purpose of this environment was to accurate model bus transactions, so that the entire FFT system, including the Avalon wrapper, could be simulated. By creating Avalon bus-functional models (BFMs) to model bus reads and writes, this environment was intended to replace the full System Level Simulation (discussed in the next section) as much as possible.

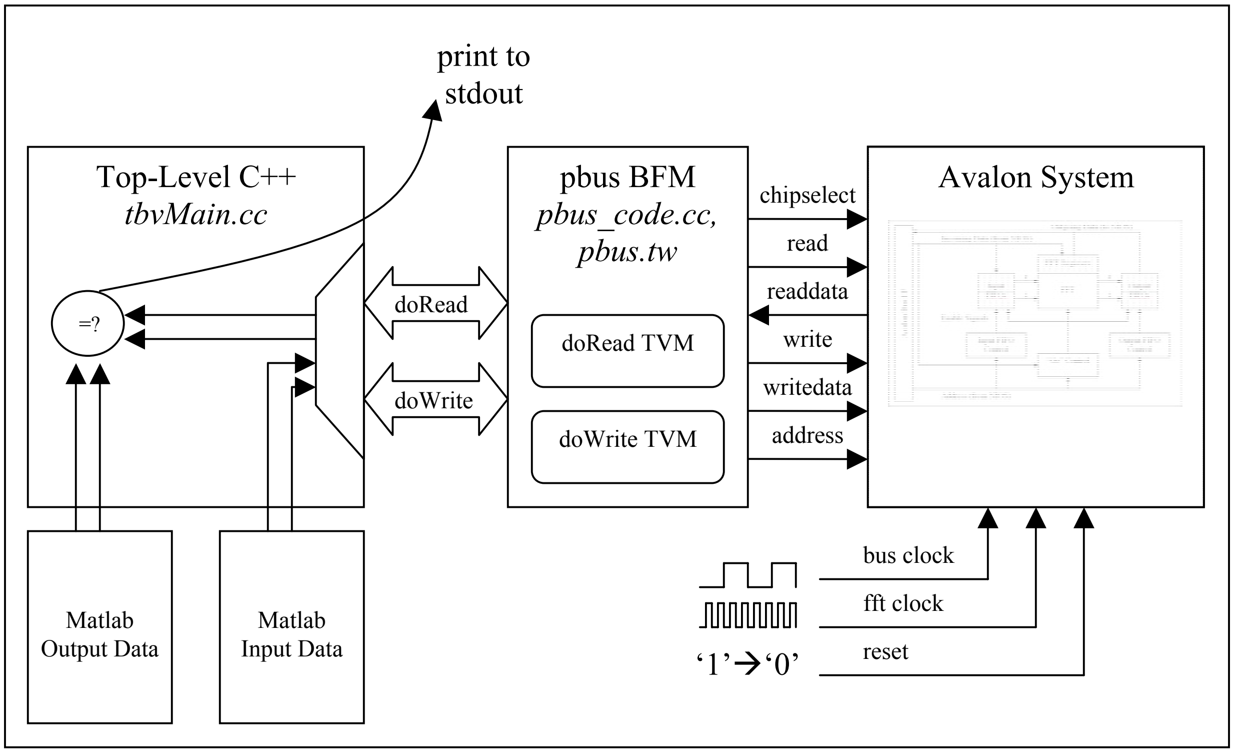

Figure 26: Complex Testbuilder Environment

Because it fully simulates the NIOS processor, system-level simulation is very resource-intensive, and wanted to be avoided at all costs.

Figure 26 shows the second Testbuilder environment. Note that in the following diagram, the Avalon model refers to the FFT and the Avalon wrapper. As previously mentioned, the Avalon wrapper includes the input FIFOs, output FIFOs, control logic to facilitate run-time data processing, and all logic to handle asynchronous clock interfacing.

6.4 Verification of the Embedded System

For the most part, Testbuilder and the Avalon bus functional models were used to verify at the system level. However, on occasion a full system-level simulation was necessary to ensure the bus functional models are accurate, and to fix bugs that were could not be caught using either Testbuilder environment. If there are were problems with the interconnection of the Avalon bus cores, or the software code on the NIOS processor, then a full simulation could be used to help determine the problem.

SOPC Builder can compile the embedded system, and set up a Modelsim project that can be used to verify the entire system. Unfortunately, by default this system is not self-checking, and the output is only waveform. Simulation of the embedded system may take days to complete, and several hundred megabytes to store waveform data. Because of this reason, full-embedded system simulation was used only as a last resort.

6.5 Hardware Verification of the Embedded System

The following steps summarize how the FFT was verified.

(1) Bit-accurate Matlab simulations are run to generate inputs and expected outputs for the FFT.

(2) The Matlab inputs are converted into an SREC file, which is then passed to the dedicated input data RAM of the embedded system.

(3) The embedded system is compiled and the FPGA is configured. At this point, the GERMS monitor is loaded onto the NIOS processor.

(4) A console is launched, and the software code fft_mem.c is compiled and downloaded to the NIOS processor. The software code is initiated, and Code execution begins and FFT outputs are displayed to the console window.

(5) The terminal output is saved, and the output is compared against the expected output, as generated by Matlab.

More details and the results of this verification are described in the Results section.

7. Results

The 16k point FFT was fully implemented as described in the above documentation. In the following subsections, we will first describe the overall directory structure and then describe how the format for FFT input and output. We then discuss the procedures for Testbuilder simulation and hardware verification. We will conclude by giving hardware statistics for two implementations. The first implementation targets the entire embedded system, which includes the FFT System, Avalon bus logic, and the NIOS processor and periphery. This implementation targets the Altera Stratix S40 FPGA, which was provided to us. It also targets a reduced FFT operating frequency of 100MHz, to show the device actually works in real-time hardware. The second implementation isolates the FFT System. It aims to meet the specifications described in Section 1.1. The implementation of a 16k-point FFT on the Altera Stratix S20 FPGA is shown to work at an operating frequency of almost 200MHz.

7.1 Directory Structure

The following describes each directory in the overall directory structure. In the following explanation, ~ is assumed to be the local working directory, checked out from CVS.

7.1.1 VHDL and Real-Time Hardware Testing Directories

~/avalon_model/fft/fft_core

The FFT System. This directory contains the top-level design as described in Section 5.1, and the components as described in Section 4.

~/avalon_model/fft

The FFT System. This directory is distinguished from the above directory, because it includes some top-level multiplexing logic, which is necessary for our testing environment, but not necessary for FFT functionality.

~/avalon_model

The Avalon bus model. The functional components of this directory are described in Section 5.2. This directory includes two subdirectories: ~/avalon_model/interface_logic and ~/avalon_model/go_logic. These two subdirectories are responsible for asynchronous clock domain and implementation of the "go" command, respectively.

~/hw_ver

The Embedded system. The functional components of this directory are described in section 5.3. Because this directory includes an Altera Quartus project, it includes many subdirectories. The most significant of these directories include ~/hw_ver/cpu_sdk/src and ~/hw_ver/ref_32_system_sim. The first includes all C source files, which will be downloaded to the NIOS processor. The second will be described below, as it is a verification directory.

7.1.2 Verification Directories

~/testbuilder_ver

This directory includes all files for Testbuilder verification.

~/testbuilder_ver/fft_core_ver

This directory includes the Testbuilder verification of the FFT System, as described in Section 6.2.

~/testbuilder_ver/avalon_ver

This directory includes the Testbuilder verification of the Avalon bus system, as described in Section 6.3. Inside this directory, there are two subdirectories: ~/testbuilder_ver/avalon_ver/cc and ~/testbuilder_ver/avalon_ver/vhdl. The first includes all C++ Testbuilder source code, as well as Avalon bus functional models. The second includes all Testbuilder VHDL code.

~/hw_ver/ref_32_system_sim

This directory includes the verification of the embedded system, as described in Section 6.4.

7.1.3 Miscellaneous Directories

This directory includes two simple C++ programs.

~/convert

This directory includes a two C++ programs. The first, input_convert.cc, converts the input data to an intermediate text file. This text file will again be converted, and then used in the input memory of the embedded system. The second, data_correct.cc, converts the Matlab output files into output files that can be compared against the hardware verification. To run both of these C++ programs, run their respective executables, named input_convert and data_correct.

~/compare

This directory includes a simple C++ program, which compares the hardware verification against the Matlab output. To run this C++ program, call the data_compare executable. Note that this executable assumes the hardware log has been saved to the file ~/hw_ver/cpu_sdk/src/hw_result.txt

7.1.4 Other Directories

~/matlab

This directory includes all the Matlab used for architectural modelling. The top-level Matlab file is ~/top_fft_compare.m. Note that this file takes several days to complete.

~/report

This directory includes the report, in several different formats.

7.2 FFT Input-Output Format

As previously discussed, the FFT uses fixed-point mathematics to process incoming data. This forced us to use a specific format for the incoming and outgoing data from the FFT module. We briefly describe it in this section.

The input to the FFT module is an 18-bit real and 18-bit imaginary stream. Each 18-bit number in this stream is in a fixed-point format, such that all but the most significant bit are considered fractional components of the number. The output is produced in the same format. To obtain reasonable results from this FFT implementation, 8-bit or less real and imaginary numbers should be used as input. The 8-bit input should be shifted such that the 8 input bits are located between bits 14 and 7. Bits 17 through 15 should be set to either 1 or 0 to accurately present the number as positive or negative in 2's complement representation.

The above implementation of fixed-point numbers restricts the data inside of the FFT to be within the range (1, -1). To extend this range we introduce an exponent that is uniformly applied to all incoming and outgoing data. This kind of fixed-point data format is commonly known as block floating point, as for a large group of numbers in the design only a single exponent is used.

To decode the data produced by the FFT unit some post-processing needs to be performed. If we assume that the incoming 8-bit data is formatted as described above then the resulting 18-bit data from the FFT should be interpreted as having an exponent of 2^{22}. This exponent value results from the fact that the FFT module introduces an exponent of 2^{12} due to data shifts between each pair of stages and the fixed-point format we use treats number 1 as $1*2^{17}$. We further multiply the exponent by 2^{-7} to reflect the 7 bit shift we performed to place the 8-bit data between bits 14 and 7 in the input.

It should be noted that deviating from the above input format specifications introduces either an overflow or additional noise into the system.

7.3 Testbuilder Verification

Recall, there are two Testbuilder verification environments. We will list steps to verify both verification environments.

1. Type
   cvs –d /pc/r/r2/vlsicourse/vlsi2004 checkout env
   in the local CVS directory
2. Type
   source env/vlsi2004.cshrc
   to set up the design environment
3. Type
   cvs –d /pc/r/r2/vlsicourse/vlsi2004 checkout fft to check out the entire FFT project.
4. Then type the following to make sure you have the most recent version of the files.
   cvs update fft
5. Enter the fft directory
   cd fft
   This location will now be referred to as the local working directory ~.
6. To verify the FFT System (as described in Section 5.1):
   cd testbuilder_ver/fft_core_ver
7. Type make to ensure all files are updated.
8. Type ./run_testbuilder to run the Testbuilder script. This will compile the C++ files and the top-level VHDL file. It will then launch Modelsim and perform verification. The verification results will be printed to the screen, and are also stored in the tbv.log file.
9. To verify the Avalon Bus System (as described in Section 5.2), from the working directory ~, type:
   cd testbuilder_ver/avalon_ver
10. Repeat steps 5 and 6 from this new Testbuilder directory. For this second Testbuilder simulation, the text of Figure 27 concludes the simulation. These lines indicate that Testbuilder simulation was indeed successful.

```
# ** Done simulation **
# ** Correct: 16384 **
# ** Incorrect: 0 **
#
# ** Closing Matlab-generated files **
#
# ** Exiting transaction-based verification **
# tbvExit was called with exitCode = 0
# Simulation is now ending.
# chpiFinish called with error code 0 and message:
# ""
# ************************************************************************
#   Testbuilder Errors/Warnings Summary
#     Errors:       0  (0 suppressed, 0 thrown)
#     Warnings:     1  (0 suppressed, 0 thrown)
#     Messages:     0  (0 suppressed, 0 thrown)
#   User generated:
#     Errors:       0  (0 suppressed, 0 thrown)
#     Warnings:     0  (0 suppressed, 0 thrown)
#     Messages:     0  (0 suppressed, 0 thrown)
# ************************************************************************
#
```

Figure 27: Testbuilder Simulation Output

7.4 Hardware Verification

The following steps may be followed to perform hardware verification. Note that these steps assume the Altera NIOS Development Kit (Stratix Professional Edition) is connected properly. That is, the UART is connected to COM1 on the host computer. As well, the JTAG is connected to LPT1 on the host computer. Finally, the board is powered up.

1. Type
   cvs –d /pc/r/r2/vlsicourse/vlsi2004 checkout env
   in the local CVS directory
2. Type
   source env/vlsi2004.cshrc
   to set up the design environment
3. Type
   cvs –d /pc/r/r2/vlsicourse/vlsi2004 checkout fft to check out the entire FFT project.
4. Then type the following to make sure you have the most recent version of the files.
   cvs update fft
5. Enter the fft directory
   cd fft
   This current location will now be referred to as the local working directory ~.
6. The following steps run the Matlab behavioural model. This step would ordinarily be necessary to compute the expected values from the FFT.

However, the expected values have already been saved in the working directory. Hence, these steps aren't necessary.

a. Enter the ~/matlab directory
b. Launch Matlab
c. Run the top_fft_compare.m script. Note that this script takes several days to complete.
d. Enter the ~/convert directory.
e. Run the input_convert C++ executable to convert the Matlab input vectors into an intermediate text representation.
f. Run the data_correct C++ executable to correctly format the Matlab output for comparison against the actual hardware verification.
g. Launch a NIOS SDK Shell (refer to the Altera documentation for more information). Traverse to the ~/matlab directory, and type the following command:
   ```
   nios-convert --infile=input_mem_unsigned.txt \
   --outfile=input_mem_unsigned.srec \
   --iformat=dat --oformat=srec --width=8
   ```
   This command converts the intermediate text representation from step e into an SREC file that can be correctly interpreted as memory contents for the input memory on the Avalon bus.

7. Using Quartus II Version 3.0 build 245, open the following Quartus II Project: ~/hw_ver/minimal_32.quartus
8. With the project open, compile the project by selecting Processing-> Start Compilation. Note that this step takes approximately 40 minutes, depending upon the performance of your host computer.
9. Click Tools -> Programmer, then program/configure the Altera Stratix S40 device.
10. Launch a NIOS SDK Shell and traverse to the following directory: ~/hw_ver/cpu_sdk/src
11. Type the following to build the most recent C file:
    ```
    nb fft_mem.c
    ```
12. Type the following to download to the NIOS processor via the JTAG connection:
    ```
    nc fft_mem.srec
    ```
    A Console window will appear.
13. Inside the Console window, type the following:
    ```
    stdio window
    ```
    An independent standard I/O window, called "Stdio" will appear.
14. Return to the Console window, and type go.
15. Return to the Stdio window, and observe the FFT outputs being printed to the screen. Once all outputs have been printed to the screen, select File -> Save Transcript, and save the transcript as the following file: ~/hw_ver/cpu_sdk/src/hw_result.txt
16. Traverse to the following directory: ~/data_compare/

17. Run the data_compare C++ executable. Follow the steps described. The script compares the Matlab output against the hardware verification output. The lines of Figure 28 conclude the simulation. These lines indicate that hardware verification was indeed successful.

```
** Done comparison **
** Correct: 16384 **
** Incorrect: 0 **
```

Figure 28: Hardware Verification Comparison Output

7.5 Final Hardware Statistics

Two implementations were performed for the FFT. The first implementation describes the FFT, along with Avalon bus interface and the NIOS environment, as described in sections 5.2 and 5.3, respectively. This implementation does not use optimize the design as highly as possible, and is performed to prove that the embedded system behaves correctly in hardware.
The second implementation only implements the FFT System, as described in Section 5.1. This implementation attempts to meet the original design specifications. The Altera Stratix S20 device is targeted, and the speed is optimized, in attempt to reach the original design specification of 200MHz.

7.5.1 Embedded System Implementation Statistics

Figure 29 shows statistics of the unoptimized embedded system, which was implemented to verify to show that the FFT works correctly in hardware.

```
Flow Status Successful - Sun Jul 04 18:16:27 2004
Top-level Entity Name minimal_32
Family Stratix
Device EP1S40F780C5
Total logic elements 13,124 / 41,250 ( 31 % )
Total pins 4 / 615 ( < 1 % )
Total memory bits 1,028,368 / 3,423,744 ( 30 % )
DSP block 9-bit elements 48 / 112 ( 42 % )
Total PLLs 1 / 12 ( 8 % )
Total DLLs 0 / 2 ( 0 % )
Bus Speed: 25MHz
FFT Core Speed: 100MHz
Maximum FFT Core Speed: 136.35MHz.
```

Figure 29: Embedded System Implementation Statistics

The results of this implementation exactly match the Matlab outputs. Hence, the following figure shows the difference between the ideal Matlab FFT and the finite-precision hardware FFT.

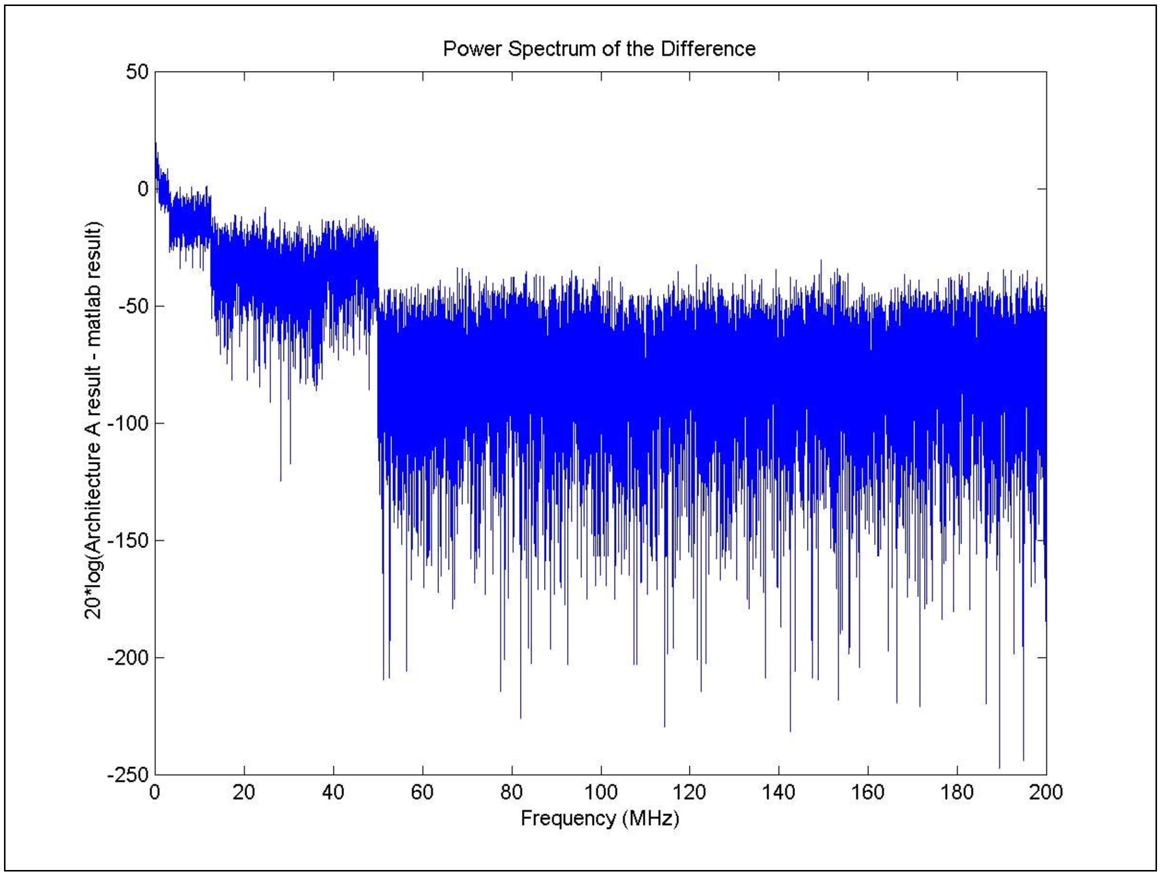

Figure 30: Power Spectrum difference between Matlab computed FFT and FFT computed by architecture B

As can be seen by Figure 30, error beyond 10MHz is less than -10dB, while error beyond 50MHz is less than -50dB.

7.5.2 FFT System Implementation Statistics

Figure 31 highlights statistics of the best possible implementation when the FFT System was targeted to the Altera 1S20 device, and optimized for speed. The optimizations used to obtain these results base on some of the compiler directives. Specifically, special attention is to be paid to the following directives:

- Preserve Hierarchical Boundary
- Auto RAM Replacement
- Remove Redundant Logic
- Remove Redundant Registers

The first directive tells the Quartus II compiler to treat a module, or an entity, as a single unit that is not to be mixed with any logic that connects to it. This option was key in separating the real and imaginary components of the complex multiplier, allowing it to run at much higher speed. The second option is used to replace the shift registers we use in the design with a RAM memory block. These two options are hard coded into the

VHDL code in various modules to ensure good final results. With these settings the design compilation in Quartus II 4.0 will report about 190MHz maximum operating frequency.

The latter two options further improve design performance by forbidding Quartus II from removing redundant logic. The redundant logic in this design primarily consists of replicated registers. These registers were duplicated on purpose to meet timing constraints and should not be removed. Thus, the latter two options should be switched off. Modifying compiler settings in the Quartus II software can do this. The final design compilation results in design performance of 199.88MHz, as shown in Figure 31.

```
Flow Status Successful - Fri Jul 02 02:21:55 2004
Revision Name fft_top
Top-level Entity Name fft_top
Family Stratix
Device EP1S20F780C5
Total logic elements 11,432 / 18,460 ( 61 % )
Total pins 206 / 587 ( 35 % )
Total memory bits 785,808 / 1,669,248 ( 47 % )
DSP block 9-bit elements 48 / 80 ( 60 % )
Total PLLs 0 / 6 ( 0 % )
Total DLLs 0 / 2 ( 0 % )
Speed: 199.88MHz.
```

Figure 31: FFT System Implementation Statistics

Further optimization is possible by using area constraints for subsections of the FFT module. Furthermore, by removing the register access capability the design performance can be improved.

8. Contributions of group members

The work for the project consisted of many tasks that were divided between all group members. While all group members performed the literature review, which was required to understand the project, other tasks were divided amongst all group members to complete the project in timely fashion. In this section of the report we give credit to the work of each group member. The contributions of Tomasz Czajkowski, Christopher Comis and Mohamed Kawokgy are summarized in the following subsections.

8.1 Contributions of Tomasz S. Czajkowski

Tomasz has completed several tasks in an effort to complete this project. These tasks are divided into five groups: Matlab modelling, Hardware Design, Integration, Vector Testing and Debugging.

8.1.1 Matlab Modelling

The Matlab modelling component of the work consisted of the generation of Matlab code that simulated the behaviour of both butterfly architectures presented in this report. These models allowed us to process sample data through our proposed architectures to determine the necessary parameter values for bit-width, N, M and f_{op}, as described in section 3.1. Because these models use finite data precisions just as our final design implementation, it was possible to determine the expected level of noise prior to final hardware design implementation.

Several Matlab scripts were also developed to automate testing of the models and compare their output to floating point precision FFT algorithms embedded in the Matlab software. The same Matlab scripts were then used by Christopher Comis and Tomasz Czajkowski to create a set of input and output vectors for hardware testing. The clear correspondence between the Matlab models and the hardware design allowed them to precisely determine the exact bit patterns that are produced by the hardware circuit.

8.1.2 Hardware Design

In the section of hardware design Tomasz was responsible for VHDL design of both butterfly architectures. This task was divided into two sections: design and optimization.

In the design part the goal was to create the hardware description to determine the resource usage and the initial speed of the circuit. The designs were implemented using a divide-and-conquer methodology. Each circuit was divided into its basic parts, keeping in mind that the goal of an effective design is that the synthesizer of the Hardware Description Language understands the intention of the designer.

These designs were then pipelined and optimized for speed in the optimization stage. It was also necessary to use specific Altera attributes, such as "PRESERVE_HIERARCHICAL_BOUNDARY," to ensure that the circuit was mapped, placed and routed such that the speed of the circuit did not suffer. The main problem here was the complex multiplier unit, which is by default mapped into the DSP block. It was necessary to force Altera Tools to not retime this portion of the circuit and keep the generation of the real and imaginary parts of the computation separately. Failing to do so caused the speed of the circuit to be lowered by roughly 50MHz.

8.1.3 Integration

The integration of the FFT module was also Tomasz's responsibility. The main goal here was to link Twiddle Factor and butterfly modules so that a single FFT stage could be computed, and then to link thus created sections of the FFT design in series to compute an FFT of 16384 points. The main challenge here was in the proper balancing of delays that governed when one stage began operation with respect to its previous stage.

8.1.4 Vector Testing

While Christopher performed most of the system level testing and design testing in the Testbuilder software, Tomasz performed the initial Vector simulations on butterfly architectures. The key contribution here was determining the parameters necessary to link the butterfly and the Twiddle Factor modules so that proper twiddle factors multiplied the output of each butterfly.

In addition vector testing at every step of integration was necessary. It turned out that it was not possible to just activate all stages of the FFT design simultaneously, but rather they had to be delayed with respect to one another. This ensured that when stage *i* of the FFT started producing correct values at its output, the stage *i+1* would behave as though it has just started operation.

8.1.5 Debugging

A considerable portion of Tomasz's time was spent on circuit debugging. During the integration of the system, both the FFT itself and the integration of the FFT into the hardware testing module, both Tomasz and Christopher spent significant amount of time tracing waveforms in the hopes of identifying the problems with the circuit. Christopher's expertise in the Testbuilder/Hardware verification environment and Tomasz's knowledge of the internal workings of the FFT module design enabled many faults to be located and remedied quickly.

8.2 Contributions of Christopher J. Comis

Chris Comis was responsible for the following tasks.

8.2.1 Avalon Bus Wrapper Logic

VHDL code was written to implement all the asynchronous clock interfacing, manipulation of the bus signals, as well as implementation of the FIFOs and the "Go" instructions.

8.2.2 Creation of the Embedded System

SOPC Builder was used to create the complete embedded system, which included the NIOS processor, the FFT, memories and a UART. For the NIOS processor, several software C codes were implemented to test the FFT at various stages in development.

(a) fft_simple.c: Simple hard-coded input data is passed to the FFT from the NIOS processor via the Avalon bus. Results from the FFT are passed from the FFT back to the NIOS, where they are printed on a host-terminal via the JTAG interface.

(b) fft_stream.c: Input data is streamed from a host-terminal window to the NIOS via the JTAG interface. Input data is then passed to the FFT via the Avalon bus. Results from the FFT are passed from the FFT back to the NIOS, where they are printed on a host-terminal via the JTAG interface.

(c) fft_mem.c: Pre-computed input data is stored in a dedicated on-chip memory, which resides on the Avalon bus. Each input data is retrieved from this on-chip memory, then passed to the FFT via the Avalon bus. Results from the FFT are passed from the FFT back to the NIOS, where they are printed on a host-terminal via the JTAG interface.

All three of these software codes were used in verification. The final (fft_mem.c) was used in the final hardware demonstration.

8.2.3 Testbuilder Verification

Both Testbuilder environments were implemented. The first Testbuilder environment was used to verify the FFT as a stand-alone system. This Testbuilder environment is relatively simple, in that it simply draws HDL signals into the C++ environment, and verifies the FFT system at the C++ level. The second Testbuilder environment verifies the FFT and all necessary Avalon bus logic (referred to as the Avalon bus model). This environment is much more sophisticated; it incorporates two transaction verification modules (TVMs) that represent an Avalon bus functional model. Both Testbuilder environments draw inputs from the Matlab simulations. Furthermore, they are both self-checking against the expected outputs from Matlab simulations. If the FFT results do not match, an error is reported.

8.2.4 System-Level Verification

Extensive system-level verification was performed, and several previously overlooked bugs were fixed. Of course, fixing these bugs at the system level proved to be very time consuming. Of the three bugs listed in Section 5.4, Bugs #2 and #3 were fixed by Chris. Bug #1 was also fixed by Chris with the assistance from Tomasz.

As well, there were several bugs in the software code and the reliability of the JTAG connection was overlooked. These bugs will not be discussed.

8.2.5 Miscellaneous

Several miscellaneous tasks were also done by Chris Comis.

- Using functions and Matlab code written by Tomasz Czajkowski, a top-level Matlab script was written to generate input and output data for the Testbuilder environments and hardware testing environment. Using a small C++ program, data had to be converted for proper comparison between the Matlab output and the output from real-time testing.
- Modelsim has tighter syntax checks than the Quartus simulator. Over 10 files had to be modified for Modelsim to compile without warning.
- Two small C++ files, data_compare and data_correct, were written.

8.3 Contributions of Mohamed Kawokgy

Mohamed's contributions throughout the course of this project involve multiple tasks and problems solving. This includes Matlab high level modelling, architectural and hardware design, functional and timing simulations and performance characterization. These tasks can be summarized as follows:

8.3.1 Matlab High Level Modelling

Matlab high level modelling was one of the earliest stages of the FFT design.

Goal: The main goal of this design stage is to gain a deeper insight into the HI spectrum and to have a clear picture of the type of FFT spectrum that we should get after the design completion. This extends to studying the effect of FFT number of points on the quality of the generated spectrum and its suitability for the astrophysics group's application of interest.

Result: The result of this analysis showed that the 32K-point FFT and the 16K-point FFT both perform almost the same in terms of the quality of spectrum generated suitable for the application of interest. This result helped relaxing the initially proposed FFT specifications from a 32-K point FFT into 16-K point FFT without any loss of spectrum information. This relaxation of the number of points helped, in turn, reducing the hardware requirements for the design and allowed its implementation on the available FPGA chip.

8.3.2 Architectural and Hardware Design

The most promising contribution of Mohamed was to propose, design and implement an ultimately new architecture for the Twiddle Factor generator module that feeds the Butterflies, at each stage of the FFT, with the corresponding Twiddle Factor. The Twiddle Factor generator module proposed by Mohamed is unique to this project and has never been done before. This module features high processing speeds as well as low hardware requirements and outperforms previously reported designs as previously discussed.

Goal: The main goals and challenges faced during this design stage can be summarized as follows:

1) The need for designing a module that can provide the Butterfly architecture of choice, namely architecture B, with the correct Twiddle Factor. The main challenge was to answer the question "*How can we design a module that generates the Twiddle Factors for each Butterfly stage of the FFT taking into consideration the irregular non-consecutive processing nature of the Butterfly legs imposed by the chosen architecture B?*" This was a considerably crucial issue since the chosen Butterfly architecture has a more irregular processing order when compared to other architectures, yet it provides a more efficient implementation, and designing a Twiddle Factor generator module to fit

such architecture wasn't an easy task. 2) The design should avoid any type of iterative processing in order to allow higher processing speeds and to meet tight timing requirements imposed by the application of interest. 3) The design should be efficient in terms of hardware cost in order to meet the tight hardware requirements, especially in terms of memory resources.

This also extends to the design of the same module with register access capabilities to allow *observability* and *accessibility* to the design.

Result: The result of this design stage was to successfully come up with a novel architecture for the purpose described above and meeting all requirements sought; simplicity of design, high operating speed and low hardware requirements. The proposed Twiddle Factor generator is flexible and can be easily integrated to the Butterfly units of architecture B as well as other regular Butterfly architectures with simple modifications which makes it an attractive architecture.

8.3.3 Functional and Timing Simulations

Intensive functional and timing simulations were performed on the designed Twiddle Factor generator module with and without register access capabilities. Multiple test benches, capable of covering all the module test cases, were used in these simulations.

Goal: The main goal of this design stage was to guarantee the proper functionality of the designed module using functional simulation. The first design and functional simulation step was performed using Modelsim tool then the design was transferred to Quartus environment. In order to measure the operating speed of the module, timing simulation was performed, after mapping the design to the FPGA of interest.

Result: The results of this stage proved that the concept of the proposed Twiddle Factor generator module is indeed promising. The functionality showed that the designed unit provides the correct Twiddle Factor to the right Butterfly of each stage at the right moment. Timing simulations proved that the operating speed of this module outperforms the speed requirements set by the application of interest.

8.3.4 Performance Characterization

Performance characterization of the final FFT design was also performed using Matlab.

Goal: The main goal of this stage was to find a numerical method to characterize and quantify the injected noise or the quantization error of the finite word length FFT model, as opposed to an ideal FFT model.

Result: In this regard, Mohamed has proposed a numerical method to characterize the quantization error based on normalization and autocorrelation concepts. By normalizing the input stream to the FFT and computing the autocorrelation of the output stream of the ideal FFT model and that of the non-ideal finite word length FFT model, the difference between these two autocorrelations gives the autocorrelation of the injected noise itself. This is since the difference between the ideal FFT output and the non-ideal FFT output is basically due to the injected noise. The value of the autocorrelation of the injected noise at zero shift (i.e. mid point of the autocorrelation sequence) gives the energy of the injected noise. The normalization is used to allow a fair comparison of the injected noise for different input streams.

9. Conclusion

In conclusion, a 16k-point FFT was implemented using fixed-point arithmetic. This FFT has an input precision of 8-bits, and an output precision of 18 bits. Internal precisions are limited by the FPGA multipliers, which are 18-bits in width.

Given the supplied input vectors, finite-precision effects produce an error of less than -10dB for frequencies beyond 10MHz, and -50dB for frequencies beyond 50MHz. The fixed-point FFT has been tested in hardware with an operating frequency of 100MHz. A separate implementation shows that the FFT may operate as a stand-alone unit at 199.88MHz.

This FFT meets the original specifications, the only exception being the reduction from 32k-points to 16k-points. This modification has only a minor effect on the overall FFT output results, and is considered acceptable.

10. References

[1] E. Chaisson and S. McMillan. Astronomy Today Fifth Edition, Prentice-Hall, 2001, ISBN 0-13-144596-0

[2] A. V. Oppenheim, A. S. Willsky, and S. H. Nawab. Signals and Systems, Second Edition, Prentice-Hall, 1997, ISBN 0-13-814757-4

[3] Altera's Home Page. Online: http://www.altera.com.

[4] G. Szedo, V. Yang and C. Dick. High-Performance FFT Processing Using Reconfigurable Logic. In *Thirty-Fifth Asilomar Conference on Signals, Systems, and Computers*, November 2001.

[5] A. Despain, Fourier Transform Computers Using CORDIC Iterations. In *IEEE Transactions of Computers*, vol. c-23, no. 10, October 1974, pp. 993-1001.

[6] E. Wold and A. Despain. Pipeline and Parallel-Pipeline FFT Processors for VLSI Implementations, in *IEEE Transactions on Computers*, vol. c-33, no. 5, May 1984, pp. 414-426.

[7] Avalon Bus Specification Manual. Online: http://www.altera.com/literature/manual/mnl_avalon_bus.pdf

A. Source Code

Please refer to the CVS Repository for all source code.

B. Suggestions for the Future

By creating this FFT project, the group has learned how to use many VLSI and FPGA CAD tools. Furthermore, the group has learned important design techniques, for medium to large-scale VLSI designs. Regardless of the overall success of this project, in hindsight the group would do several things differently, mostly in the areas of verification. These suggestions for the future are summarized below.

1. Even small components must be fully verified

Most of the bugs at the system-integration phase were due to inadequate verification at the component level. For example, verifying the Enable signal of components would have solved Bugs #2 and #3 (refer to Section 5.4 for more information). These bugs are very easy to solve at the component level, and very difficult to solve at the system level.

2. An identical toolset must be used by all group members

The integration process was also very difficult because group members used different tools for design and component verification.

As a first example, the Altera Quartus simulator was used for verification at the component level. However, Modeltech Modelsim was used at the system-level. Because Modelsim is arguably a stronger simulator, using it at the component level would have caught many bugs found at the system level. Furthermore, using Modelsim at the component level would have also allowed some of the component-level verification suites to be reused at the system-level. This idea of verification-reuse could also be extended to the reuse of Testbuilder modules.

As a second example, one group member used Mentor Graphics' HDL Designer to generate VHDL code for components. This tool is useful as a graphical visual aid. However, in creating the VHDL code from the graphical user interface, all signals are labelled with the prefix SYNTHESIZED_WIRE. For example,

```
signal SYNTHESIZED_WIRE_6     : std_logic;
```

This generic signal naming convention made it very difficult for other group members to debug code.