To the Graduate Council:

I am submitting herewith a thesis written by Adam Robert Miller entitled "Development and Verification of Parameterized Digital Signal Processing Macros for Microelectronic Systems." I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Electrical Engineering.

D.W. Bouldin

D.W. Bouldin, Major Professor

We have read this thesis
and recommend its acceptance:

Gregory D. Peterson

Chandra Tan

Accepted for the Council:

Anne Mayhew

Vice Provost and Dean of
Graduate Studies

(Original signatures on file with official student records)

Development and Verification of Parameterized Digital Signal Processing Macros for Microelectronic Systems

A Thesis
Presented for the
Master of Science
The University of Tennessee, Knoxville

Adam Robert Miller
August, 2003

Acknowledgments

First, I would like to thank Dr. Don Bouldin for providing me with the opportunity to do this research. This research reinforced the teachings of my graduate classes, and in most cases went more in-depth on some of the topics and ideas. Secondly, I would like to thank Dr. Chandra Tan and Mr. Fuat Karakaya for assisting me with my research. Without Dr. Tan's knowledge of the intricacies of the software tools involved, a lot of the development time would have taken longer, if it was able to be done at all. Thirdly, I would like to thank the Electrical and Computer Engineering Department at the University of Tennessee for supporting me with a Graduate Teaching Assistantship until I was able to transition into this research project.

Most importantly, I would like to thank my wife Lara for her support. Without her, I do not know if I would have gone into the Master's Degree program in the first place. It was with her support and understanding that I was able to complete this thesis.

This work was partially sponsored by the Defense Advanced Research Projects Agency and the Air Force Research Laboratory under agreement F30602-01-2-0562.

Abstract

Digital system design is a broad field that is growing every day. As technology grows, the complexity of systems grows also, which leads to longer design times. A Design-for-reuse policy can decrease design time by building flexibility into designs as they are created. By creating parameterized macros, they are more likely to be reused. Verifying the capabilities of macros is also important, and testing should be incorporated into each step of the design process. In this thesis, designing parameterized macros is discussed, with a Complex Fast Fourier Transform presented as an example of a complex algorithm, and three different Rounder blocks as examples of simple macros. Each Rounder was tested successfully, instantiating one of the Rounders with several different configurations. The Fast Fourier Transform macro was simulated successfully to Post-layout Simulation for one set of parameters, and for several sets was simulated in the Pre-synthesis step. The Fast Fourier Transform macro was fabricated using a TSMC 180nm process and verified to be working correctly.

Table of Contents

Chapter Page

1. **Introduction** **1**
2. **Background** **3**
   - 2.1 VHDL and Design Flow 3
   - 2.2 Design-for-Reuse 5
   - 2.3 DSP 7
     - 2.3.1 Overview 7
     - 2.3.2 The Fast Fourier Transform 9
     - 2.3.3 Rounding 10
   - 2.4 Verification 11
     - 2.4.1 Overview 11
     - 2.4.2 Simulation Types 12
     - 2.4.3 Testing 13
3. **Implementation** **16**
   - 3.1 Rounders 16
     - 3.1.1 Fixed Rounder 16
     - 3.1.2 Configurable Rounder 18
     - 3.1.3 Output Gain Stage 19
   - 3.2 FFT 21
     - 3.2.1 Shift Registers, Adders, and Subtract Modules 22
     - 3.2.2 Butterfly Types 22
     - 3.2.3 Complex Multiplier 25
     - 3.2.4 Twiddle Factors 26
     - 3.2.5 Control Logic 27
     - 3.2.6 Structure 27
     - 3.2.7 Test Bench 31
4. **Results** **33**
   - 4.1 Rounders 33
     - 4.1.1 Fixed Rounder 33
     - 4.1.2 Configurable Rounder 35
     - 4.1.3 Output Gain Stage 35
   - 4.2 FFT Results 38
     - 4.2.1 MATLAB 38
     - 4.2.2 Pre-synthesis and Pre-layout 38
     - 4.2.3 Layout and Post-layout 40
     - 4.2.4 Hardware Testing 40
     - 4.2.4 FFT Flexibility 44
5. **Conclusions and Future Work** **48**

List of References **49**

Appendices **51**

Vita **106**

List of Figures

Figure 2-1: Digital Circuit Design Flow 4
Figure 2-2: Butterfly Structure 10
Figure 2-3: BIST Structure 14
Figure 3-1: Fixed Rounder Algorithm 17
Figure 3-2: Configurable Rounder Structure 18
Figure 3-3: Rounder Macro Hierarchy 20
Figure 3-4: Output Gain Stage Structure 20
Figure 3-5: Butterfly Structures in Radix2^2SDF Algorithm 23
Figure 3-6: Complex Multiplier Structure 25
Figure 3-7: Top Level FFT Generation Flow 28
Figure 3-8: FFT Stage Structures 29
Figure 3-9: Generated Structure of 64-point FFT 30
Figure 4-1: Sample of Fixed Rounder Test Bench Simulation 34
Figure 4-2: Sample of Configurable Rounder Test Bench Simulation 36
Figure 4-3: Sample of Output Gain Stage Test Bench Simulation 37
Figure 4-4: Pre-synthesis and Pre-layout Simulation Sample 39
Figure 4-5: Layout of the 64-point FFT 41
Figure 4-6: Post-layout Simulation Sample 42
Figure 4-7: Layout of Fabricated Chip 43
Figure 4-8: FFT Simulations for Different N: 8, 32 45
Figure 4-9: FFT Simulations for Different N: 64, 128 46
Figure 4-10: FFT Simulations for Different N: 256, 1024 47

Chapter 1: Introduction

Digital design is the process in which an algorithm is created in a digital system. As technology advances to create smaller and smaller transistors, the capabilities of digital systems grow. Digital systems are increasing in speed and capacity, and are decreasing in power. This opens up new markets for digital systems. For example, cellular phones with the ability to take pictures or record movies are a result of the digital systems in those components becoming more advanced.

As the capabilities of digital systems increase, more complex algorithms and designs can be created. This increasing complexity leads to a longer design time. Design time can be shortened to some extent by allocating more engineers or other resources to a project. This can be costly, however. If there is competition in a design's target market, longer design time can lead to loss of sales to a competitor. If a design is new or innovative, a shorter design time can lead to more sales before any competition enters the market. Thus, increasing design time is proportional to increasing costs.

One way to reduce costs is to build on existing designs. This can be hard to do, as the design might not have been initially created with the ability to build onto it in mind. Creating designs so the can be reused in later projects can solve this problem. By designing components to be reused, the design time decreases significantly. Making a design more flexible is one way to make it reusable. If the characteristics of an algorithm can be assigned specific parameters, it can be designed with the intent to make those parameters variable. Ideally, an entire library of parameterized macros would be created and added to.

Verification is important to the design process. Verification is a broad term describing all the testing that is done on a design. By testing at different points in the design process, errors are caught earlier and corrected. By catching errors earlier, costly mistakes can be avoided.

Digital signal processing is the study of real-world information that has been converted to a signal that a digital system can understand. Signals are gathered and then converted from an analog source to digital. This process is called sampling. By sampling signals fast enough, it is possible to represent them in a digital system with no loss of information. Signals that are sampled are in the time domain; they represent a point of data on a signal at a certain time.

One of the fundamental operations performed in digital signal processing is to convert a time domain signal to a frequency domain signal. A point in a frequency domain signal

represents the magnitude of a sine or cosine wave at that frequency. One type of algorithm that is used to convert between the domains, and is optimized for use in digital systems, is called the Fast Fourier Transform. This type of algorithm has many variations.

The work for this thesis was done as part of a research project with the Defense Advanced Research Projects Agency (DARPA) and Boeing. The research involved exploring ways to improve the power, delay, and area of application specific integrated circuits. Several macros were created to test, so all the groups involved would be able to compare their improvement methods based on the same code. Boeing provided the specifications for all the macros, and test benches for some of them, like the Rounder macros.

The goal of this research is to design and verify a parameterized Fast Fourier Transform macro as an example of the process used to create and test a parameterized design. The Fast Fourier Transform is used because the algorithm is moderately complex and it is used in a wide variety of digital signal processing applications. Several types of Rounders are also created as simple examples of a parameterized design.

By using the Fast Fourier Transform, this research will show that a complicated parameterized macro can be created and verified. It will show the range of flexibility that can be created. Verification will be performed in several stages of the design process. The design will also be fabricated and tested, completing the last step of verification. By creating the rounder macros, a simple parameterized macro is presented that is easier to understand than the Fast Fourier Transform macro.

Chapter 2: Background

2.1 VHDL and Design Flow

VHDL is a programming language that is used to describe hardware. It stands for VHSIC (Very High Speed Integrated Circuit) Hardware Description Language [1]. VHDL descriptions can be synthesized into field programmable gate arrays (FPGAs) and application specific integrated circuits (ASICs). FPGAs are commonly designed to be reprogrammable, so they are often used to test algorithms. ASICs are chips that are designed with a specific purpose in mind, and are generally not reprogrammable, but they are usually faster than FPGAs.

Figure 2-1 shows a basic flow for designing digital circuits. First, specifications for the design are created by analyzing the requirements of the problem. Then code is created to fit the specifications. In this example, the code is written in VHDL and tested to see if it is functionally correct. Once the code is functionally correct, it is synthesized. Synthesis is the process of translating the VHDL code into gates and logical functions of a specific technology. If the target technology is an FPGA, the hardware resources on the FPGA are allocated to the different logic functions of the code. However, the hardware resource assignments are not specific yet. For example, a hypothetical FPGA might have 2000 NAND gates available, and when a design is synthesized, it might require 50 NAND gates. The synthesis tool checks to see if 50 NAND gates are available, and if they are, it notes that 50 NAND gates have been used, but not which 50 were used. If the target technology is an ASIC, synthesis involves taking different logic functions from a library of standard cells. The standard cell library is created with cells that have different logic functions, and sometimes variants of the same logic function that are designed to handle higher loads on the outputs. The synthesis tool keeps track of which gates are used to synthesize different functions from the code. The design is tested again after it has been synthesized, and if it is correct, a listing of the logic functions and their connections, the gate-level netlist, is passed to the placement and routing step.

After the design has been synthesized, the hardware resources from the previous step need to be placed, and the connections between those resources routed. Placement and Routing, or PAR, involves attempting to increase the frequency the design can run at by decreasing the longest delay, or critical path. Some other goals during PAR can be decreasing the total hardware

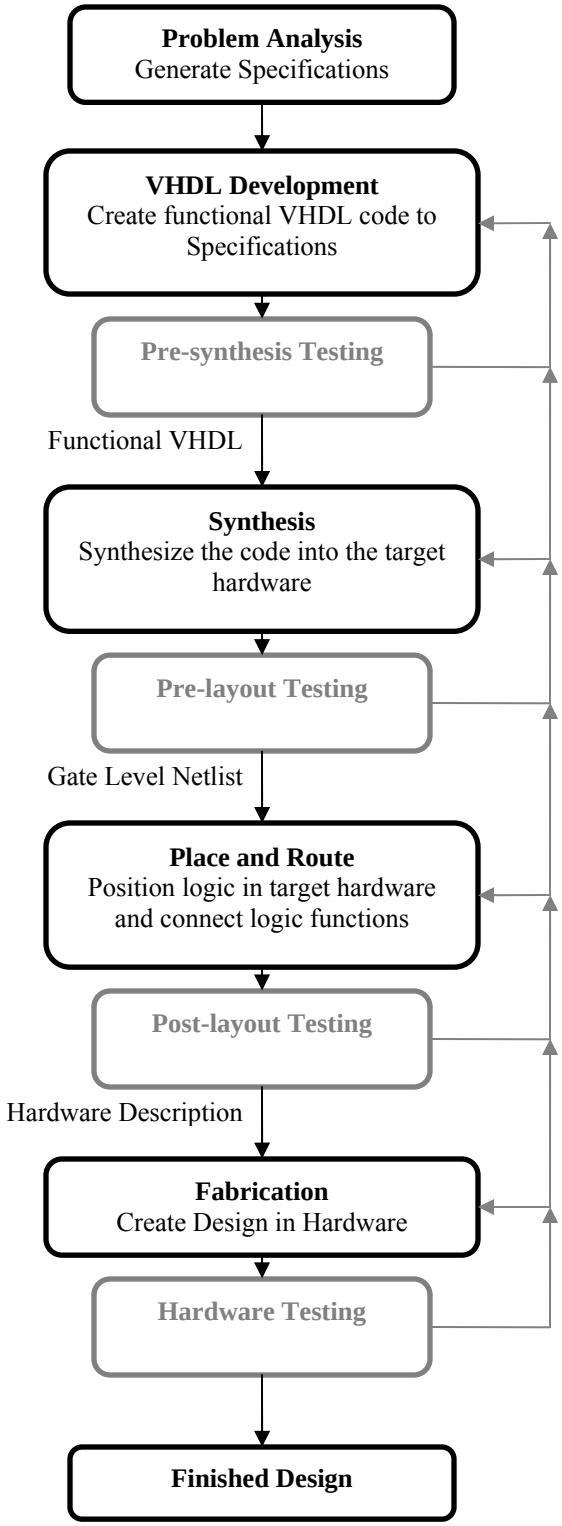


Figure 2-1: Digital Circuit Design Flow

resources used, decreasing the power use, or some combination of the three. After PAR is finished generating the layout, and the layout is verified to be correct, the design can be realized in hardware. For FPGAs it is the process of configuring the FPGA with the design, which usually takes less than a minute. Fabricating an ASIC involves creating the design in silicon, and can take two months or more from the time the design is sent to the fabrication company till the chips are received by the designer.

The most important steps in the design flow are represented by the gray arrows and boxes in Figure 2-1. These highlight the testing and verification process. Between each step in the design process testing should be done to determine if the design is functioning correctly, based on the specifications. Testing can lead to changing the previous step or possibly going back several steps in the design process to correct an error that was not caught at earlier levels. The box labeled "Hardware Testing" represents the final tests done on the fabricated design to verify that it was fabricated correctly and there are no errors in the design. Only after the hardware is tested is the design process complete.

2.2 Design-for-Reuse

Design-for-reuse is a method of developing algorithms so they can be reused in other projects. This has the advantage of saving design time in later projects, but can cost more time in the initial development of the algorithm. By saving design time, it can reduce the number of engineers needed per project, speed up the time to market, and reduce expenses. One of the ways to design algorithms for reuse is to create code that is reconfigurable. The term "macro" is sometimes applied to a block of code if it is reconfigurable.

The more flexible the code becomes, the more likely that the code will be able to be reused. However, code that is highly flexible may be more complicated for a new user. This relationship leads to a balance between flexibility and ease of use. One way to keep the code easy to use is to provide default settings for some of the things that can be reconfigured. A user that is more familiar with the algorithm can then go in and modify these default settings for an application that requires it.

In VHDL, there are two primary ways to create reconfigurable code. The first is using the **generic** declaration [2]. **Generics** can be described as constants that can change with different instantiations of the same block of code. In other words, **generics** are a way of specifying parameters for a given instance of a block of code. For example, a counter could be

created using a **generic** that stopped at a certain value. Each time the counter block is used, a different stop value can be passed to the block. Another example is a shift register with variable numbers of stages. The number of stages could be passed to the block of code, so the same block of code could be used any time a shift register is required. To make it even more flexible, the width of the data being sent into the shift register could also be assigned as a **generic**. The following code is the declaration of a shift register using **generics**. It is designed to perform a

```
entity shiftregN is
      generic( data_width : integer :=25;
      n : integer :=4);
      port( clock, resetn : in std_logic;
      write_data : in std_logic_vector(data_width-1 downto 0);
      read_data : out std_logic_vector(data_width-1 downto 0));
```

shift whenever the `clock` is rising and the `resetn` is high. Note that the width of `read_data` and `write_data` is specified by the generic `data_width`. Because `data_width` and `n` are assigned default values, it is possible to skip mapping **generics** to the block when it is instantiated in a higher-level function, and just use the default values.

The second way to create reconfigurable code is to use the **generate** statement. The **generate** statement is handled during synthesis, and can simplify the logic needed for an algorithm. There are two types of **generates** that can be used to create flexible code. The first is called a "conditional **generate**". This type of statement will allow logic to be created during synthesis if a certain condition exists. For example, the following code is designed to

```
assign_a: if a_in=1 generate
      Y<=A;
end generate assign_a;
assign_b: if a_in/=1 generate
      Y<=B;
end generate assign_b;
```

assign an input signal, `A`, to the output signal `Y` if `a_in` is equal to 1. A second conditional **generate** is included to assign `B` to `Y` if `a_in` is not equal to 1.

There are many circumstances where conditional **generates** can be used. For example, an algorithm might call for the ability to add or remove pipeline registers before synthesis, to test timing. Conditional **generates** could be applied to this requirement so that the only difference in the code between the pipelined and unpipelined version is one **generic** assignment. The main advantage to using conditional **generates** is that they allow for different hardware options when designing an algorithm. Unused options will not be synthesized, lowering the amount of resources required compared to a version that includes redundant logic.

The second type of **generate** statement involves a loop. This type of **generate** is useful when an operation needs to be repeated, either in parallel or in series. Examples of this include adder trees and multipliers. Loop **generates** can also be used within conditional **generates**, and the reverse works as well. By combining them with **generics**, it is possible to create code that is flexible, and is more likely to be reused in future designs. More examples of code with **generic** and **generate** statements can be found in *The Designer's Guide to VHDL, 2nd Edition* [2], or online at "The VHDL Language Guide" [3].

2.3 DSP

In this section, an overview of Digital Signal Processing, or DSP, will be given, and background information on two different aspects of DSP will be presented. The first is the Fast Fourier Transform, and how it works in general. Information about the specific algorithm developed will be presented in the Implementation chapter. The second aspect that will be presented is rounding. This aspect is presented because the theory behind it is easy to understand, and it makes a good example of a simple parameterized algorithm.

2.3.1 Overview

Digital Signal Processing is used in a wide variety of applications. Some examples include voice recognition software, automatic target recognition for weapons, and cellular phones. Any time a signal from the real world is analyzed as a digital representation of that signal, some form of DSP is being applied. DSP can be performed in the time domain, taking the values of the signal and manipulating them, or the signal can be converted to the frequency domain, and processing can be done there. Both have different advantages, and converting between them is usually done with a Fast Fourier Transform.

Digital signal processing has its roots in analog signal processing, but instead of using numbers to represent signal values, the actual signals are used in analog signal processing. The

equations and filtering techniques of DSP are circuits made from resistors, capacitors, inductors, and operational amplifiers. For example, band pass filters can be created using operational amplifiers and some resistor and capacitor networks. In analog signal processing, this filter has a certain range of signal frequencies that it will allow to pass through with little to no change, and as the signal frequencies get farther away from the passed band, they are decreased. A DSP version of the same filter is ideal. It is possible to control the pass band, and remove any signals not in the pass band, instead of decreasing them gradually. However, there are two fundamental problems with digital signal processing: quantization and sampling.

Both quantization and sampling problems are introduced when the analog signals being processed are converted to digital. This conversion is referred to as analog to digital conversion, or ADC. Typically, ADC is done on voltage levels, but it can be applied to other analog signals. During the conversion process, the strength of the input signal is compared to the maximum input value, and assigned a number. For example, if you had an ADC that took an input signal from 0 to 5 volts, with an output range of 0-255, and a 3.20-volt signal were the input, the output would be 163.84. However, the output is limited to whole numbers, so it would either be 163 or 164. The potential for two different results is the quantization error. A way to minimize this error is to increase the output range of the ADC, which is equivalent to increasing the bit-width of that digital signal. Using the same example with a 12-bit ADC (output range of 0-4095) gives a result of 2621.44. There is still an error, but the error is smaller by a factor of 16, because the 12-bit ADC can detect much smaller quantum differences in the signal than the 8-bit ADC.

Sampling problems are related to how fast the ADC can run. How often the ADC outputs the converted signal value is called the sampling frequency. Because the signal is not continuous anymore, there is a limit on the maximum frequency that can be represented by the digital signal. To accurately represent the signal, there cannot be any data in the signal that is at a frequency higher than half the sampling frequency. This limit is called the Nyquist frequency [4]. Sampling at higher frequencies can solve this problem.

Once a signal is converted from analog to digital, there are two different domains to process the signal. The first is the time domain. Signals in this domain have different magnitudes at different times. This is how we perceive all signals. All DSP begins with signals in this domain, and they can be converted to the frequency domain and back again. In the frequency domain, signals are represented as different magnitudes at different frequencies [5].

There are advantages and disadvantages to processing in each domain. One of the advantages of time domain processing is the ability to further sample the signal. Windowing, or allowing the signal to pass for a certain amount of time, is also easy to accomplish in the time domain. However, it is limited in its applications. Many types of filters are easier to realize in the frequency domain. Simple low pass filters are much less complicated in the frequency domain than the time domain. Correlation is also much easier to determine in the frequency domain. The key disadvantage of the frequency domain is that it deals with blocks of samples instead of a continuous stream of samples.

2.3.2 The Fast Fourier Transform

The Fourier analysis, named after Jean Baptiste Joseph Fourier, is a process in which a signal is broken down to sinusoidal waves, with varying frequencies and amplitudes. This is done to make it easier to perform mathematical operations on the signal. There are two different traits of the signal that set how the signal will be processed. If a signal is periodic, meaning it will repeat to infinity, and continuous, then the type of Fourier transform that is performed on it is called the Fourier series. If the signal is not periodic, but still continuous, then the transform is called the Fourier Transform. Sampled signals that are not periodic are transformed using the Discrete Time Fourier Transform. Finally, signals that are sampled and periodic are processed using the Discrete Fourier Transform or DFT [4].

The DFT can be expressed as a simple equation. In this equation, k runs from 0 to $N/2$. This form of the DFT only shows the positive half of the frequency range, but it is commonly

$$X[k] = \sum_{i=0}^{N-1} x[i]\cos(2\pi ki / N) - \sum_{i=0}^{N-1} x[i]\sin(2\pi ki / N) \quad (1)$$

represented as the range from $-N/2$ to $N/2$. N is the number of samples being taken. Directly implementing this formula would result in an execution time that is proportional to the number of samples squared.

The Fast Fourier Transform, or FFT, is an algorithm that simplifies the DFT. By processing x in a different order and breaking the operation down into the structure shown in Figure 2-2, it is possible to calculate the FFT with an execution time that is proportional to $N\log_2(N)$. The structure shown in Figure 2-2 is called a butterfly, and it is the basic building block of most FFT algorithms.

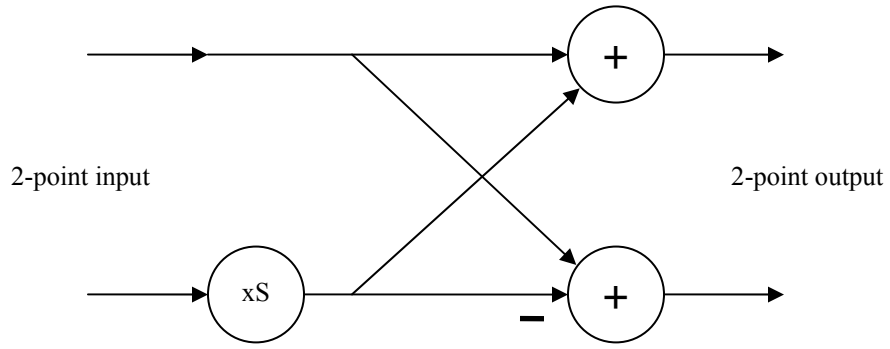


Figure 2-2: Butterfly Structure

Because of the way the FFT is structured, it can be used to convert from a frequency domain signal to a time domain signal. This is called the Inverse Fast Fourier Transform, or IFFT. To convert it, the frequency domain signal is run through the FFT algorithm. Then, the samples from n=1 to n=N-1 are mirrored around (N-1)/2 [6]. The sample at n=0 is unchanged. By reversing most of the signal, it has converted from an FFT to an IFFT.

2.3.3 Rounding

Rounding is also important to DSP. Some operations produce outputs that are significantly larger than one of the inputs. Multipliers, for example, produce an output with a bit width equal to the sum of the input bit widths. Connecting several of these types of operations together can lead to very wide signals. This can lead to the use of large amounts of hardware resources. One way to compensate for this is to truncate the signals to a smaller size. However, this reduces the accuracy of the data. A balance between the two is to round the data off instead of truncating it. Rounding in binary is much simpler than rounding in decimal. In binary, if the signal is positive and the bit to the right of the rounding point is one, then it rounds up. What makes this simpler in binary is the check to see if the digit above the rounding threshold is removed. By adding the bit to the right of the rounding point to the bits to the left of the rounding point, a positive number will be rounded.

Sometimes a certain level of accuracy is required, and some lower bits of data cannot be lost. In this case, rounding is not an option. Instead, the higher level bits are cut off. A contingency is put in so that if the signal is too large for the remaining bits to represent, it will set

the signal to the maximum value it can represent. Combining this with a rounder leads to a very flexible block of code that can be used for either rounding or truncating depending on the situation.

2.4 Verification

In this section, verification and testing will be discussed. After presenting an overview of verification, different simulation types will be discussed. Finally, testing will be discussed, with the focus on having test circuitry as part of a fabricated design versus having separate test circuitry.

2.4.1 Overview

Verification is an important part of system design. It is the process of verifying the functionality of the system at different stages of the design process, usually by providing some functional input to the design and comparing the output to the expected output. Testing and verification can reduce costs in designs by reducing the chance that some or all of the design will not work when fabricated. By testing at many different levels of the design process, it is easier to isolate where problems are introduced and correct them. Verification should place the design under stress, so that it can be shown to be working for most of the combinations of parameters. By testing to the limits of the specifications, it is more likely that problems throughout the range of the specifications will be caught.

There are two different ways to verify that a design is working as intended. The first is to create a model of the system in a higher-level language that will produce bit-true results. This has the advantage of matching the output from the design exactly. It may also be possible to create the model so that internal values are bit true also. This is very helpful in debugging the design. However, developing a bit-true model can take a long time, depending on the algorithm and how it was originally created.

The second method of verification is to create an approximate model, or use an existing model that will produce similar results. This has the advantage of being easy to develop, and can require less time to develop, because it is not an exact model. In the case of DSP, there are existing tools that can duplicate some algorithms very easily and quickly. The disadvantage is that the output from the design must be compared to the output from the model, and they might not exactly match. It must then be determined, by looking at the outputs and the algorithm, if the design is working as intended.

2.4.2 Simulation Types

When performing tests on the design, there are two different options that are available at most stages of the design process. The first is an event driven type of simulation. In this type of simulation each operation is assumed to take a certain quantum delay, Δ. For example, if a process assigned a value to a signal on the rising edge of the clock, and the clock edge occurred at 100ns, then the output would change at 100ns plus 1Δ. This type of simulation can be fast, but it is not accurate for timing, only for logic. The second type of simulation is much more complicated. It can only be performed after the system has been synthesized. It uses models of the gates involved and SPICE, a circuit simulation tool, to calculate what the voltages are for every net in the system. This type of simulation can be more accurate, but because of the number of calculations involved, it usually takes more time to perform. A SPICE simulation can take ten times longer or more to perform.

Simulation should be performed at each stage of the design process. Doing so makes it easier to go back to a higher level of abstraction to correct an error. As the design becomes less abstract, the simulation must become more accurate. A purely event driven simulation would not be useful after the layout is generated, because it would not take into account the layout itself. Also, an approximation type of simulation should not be used past the Pre-synthesis testing stage. The results of the Pre-synthesis testing can be used to compare with the test results of the later stages.

At the Pre-synthesis stage, an event driven simulation can be useful to determine if the algorithm has any errors. If the test vectors and expected output are approximate instead of bit-true, the outputs from the design at this stage should be used in later simulations to confirm that they are accurate. It should be noted, however, that because there is no timing information at this stage in the design process, intermediate values can be inaccurate when compared to values in later stages. The stable results should be correct, however.

After the design has been synthesized, it is known what logic blocks will be used to create the algorithm. In ASICs, these logic blocks can be standard cells. In FPGAs, these blocks are usually specific to the hardware being targeted. At this level, an event driven simulation with timing information can be performed. This is similar to the previous stage, but instead of assuming a Δ delay for each operation, the delay of each gate is known according to the load on the gate. This timing information can be calculated based on the SPICE model of each gate and the load characteristics it will be under. This produces a more accurate test. Testing is done at

this level because it is possible to introduce timing errors when the delay for the gates has been introduced into the system. For example, in a system where some combinational logic is between two registers, there might be too many gates between the registers, causing an error to occur. This level of simulation does not take into account the wire delays, because while the type of hardware is known, the position of the gates relative to each other is unknown. This will introduce more delay, and is included in the next step. It is also possible at this step to perform simulations using a fixed timing value for each gate. By not calculating the delays of each gate, time is saved at the cost of accuracy of the simulation. If the timing for the gates is estimated to be higher than the actual delay and there are no problems with the simulation, it is likely that the system is still working correctly.

Simulation after the layout has been generated is the most important simulation to perform. At this stage, the logic required and its locations are known. This means that the length of the wires and the delay that they introduce to the system is also known. As technologies decrease in feature size, wire delay is becoming more of a problem. At this level, it is still possible to calculate the delay of the gates and the wires connecting them, making the event driven simulation with delay information a good way to simulate the system. SPICE simulation is possible, but it can take a long time to generate the net list, and takes a long time to run through each simulation. If the delays are calculated, which is called back annotation, the input stimulus can be changed easily and the simulation can be run faster than a SPICE simulation. While SPICE will give the most accurate results, performing an event driven simulation with timing information will give a good estimate for significantly less time involved.

2.4.3 Testing

After creating a design in hardware, either through fabricating an ASIC or programming an FPGA, it is necessary to test the design to verify that it is working. Ideally, the behavior of the post-layout simulation is very close to what the hardware will be, so any design problems the hardware would have had are already solved. Testing can either be done with hardware in the chip, or by test equipment outside the chip.

One way to put test equipment on the chip is with a built in self-test, or BIST. Figure 2-3 shows an example of a BIST. In order to use the BIST, a set of test vectors are chosen and programmed into the ROM. After the test vectors are chosen, they are used with the design under test and the signature compressor to generate a signature. The signature is the output of the signature compressor after all the test vectors have been input. An alternative to using a ROM

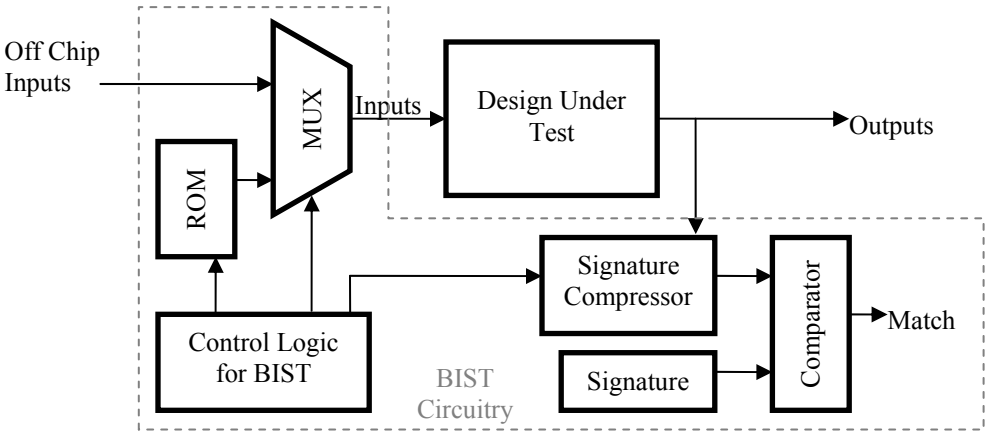


Figure 2-3: BIST Structure

and test vectors is to use a counter or other type of number generator. Signature compression is an algorithm that retains a memory of the previous compressor outputs and performs some XOR operations on the previous output and the next input to the signature compressor. Since it has feedback, it is only done on the rising edge of the clock. Additional XOR gates are inserted between different outputs, so with no input it will output a continuous stream of numbers and will not repeat any until it has cycled through all the possible outputs. This type of signature compressor is called a Linear Feedback Shift Register or LFSR [7].

After the signature has been generated in simulation, it is programmed into the VHDL for the BIST. When the BIST is run, it feeds the test vectors into the design under test and compresses the output in the signature compressor. After the test vectors are finished running, it compares the output of the signature compressor to the simulated signature. If they match, the comparator outputs a signal that acknowledges they match.

There are several advantages to having the testing circuitry as part of the design. One advantage is that it allows the test mode to be simple, with minimal external hardware required. This advantage is important in a research or teaching environment, where testing equipment for specific hardware may be too expensive. The second advantage is that the testing hardware runs at full speed. Instead of having delay going on and off the chip and delay caused by any circuitry from external test equipment, there is very little delay between the testing circuitry and the design under test.

There are drawbacks to this approach also. While there is very little delay between the test circuitry and the design under test, the test circuitry adds some delay to the overall design. This slows down the overall performance of the chip. The second disadvantage is in having extra circuitry on the chip. Having the extra circuitry on the chip takes up more space and the power consumption increases.

If space, power, and speed cannot be sacrificed, another way to test a design is to put the test circuitry off the chip. In addition to decreasing the space and power required for the design, it allows the user to create new test vectors and input them into the circuit. Depending on the complexity of the testing hardware, the test board can be expensive. Running extra test circuitry can slow down the design, because of the time required to get the test vectors onto the chip and the outputs off of the chip.

Chapter 3: Implementation

3.1 Rounders

There are three different rounder blocks that have been developed for this paper. They are being presented as an example of parameterized code. The lowest level is called the Fixed Rounder. Additional hardware is added to the Fixed Rounder to create the Configurable Rounder. The third rounder also performs some shifting on the data and is called the Output Gain Stage. Code for the Rounders is included in Appendix A.

3.1.1 Fixed Rounder

The Fixed Rounder block is designed to take an input (`d_in`) of `input_width` bits wide. It then is rounded at the radix position set by the `fr_rad_pos` parameter. After rounding, the output is set to be the lowest `output_width` bits in the rounded number. If the number is too large to be displayed in `output_width` bits, the rounder will set the value to the maximum. Normally negative number can reach a value of (`output_width`-1)^2, but the Fixed Rounder will limit the output magnitude to (`output_width`-1)^2-1. Figure 3-1 shows the logic for the Fixed Rounder. The output of the Fixed Rounder is `q_out`.

In addition to the flow shown in Figure 3-1, other parameters are used to specify whether or not there is a pipeline register on the input (`ri`), what kind of register is on the output (`ro`), and what type of reset the registers use (`rt`). When `ri` is zero, there is no register on the input and when it is one, there is a pipeline register on the input. The pipeline register inserts a delay of one clock cycle, and how it is reset is controlled by `rt`. If `rt` is 0, there is no reset. When `rt` is 2, the reset is asynchronous. The default value for `rt` is 1, and has a synchronous reset then. For the output register, `ro` has the same behavior as `ri` if `ro` is zero or one. It can also be set to two, which will insert a register that only outputs on the rising edge of the clock and when the `out_load_enable` signal is set to high. This type of register is referred to as a "z register." If `ri` and `ro` are zero, then the `clock`, `reset_n`, and `out_load_enable` signals are not needed.

Once the fixed rounder is synthesized, all the parameter values are fixed. For example, if a Fixed Rounder block had `fr_rad_pos` equal to 4, `input_width` equal to 12, and `output_width` set to 10, it would take the path shown by the dotted line in Figure 3-1. The

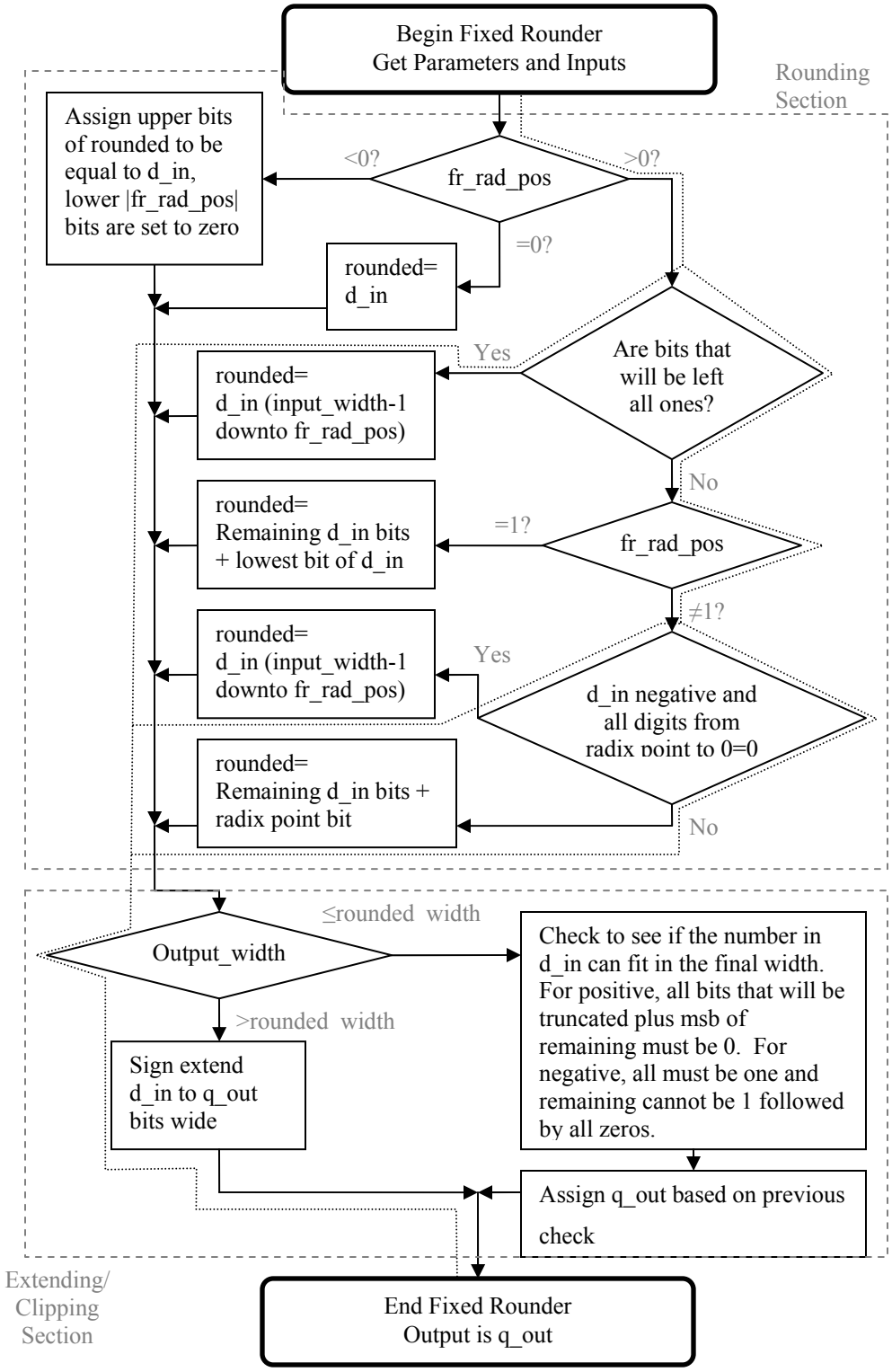


Figure 3-1: Fixed Rounder Algorithm

line can take several different routes, depending on what the value of `d_in` is. Logic not on the path shown will not be synthesized, because it is redundant. This means that each time the Fixed Rounder is instantiated, it will only round a specific bit width at a specific radix point, and will only extend or truncate to a fixed output width.

3.1.2 Configurable Rounder

The Configurable Rounder expands on the Fixed Rounder. By creating an array of input signals to be selected, it allows for the radix point to be adjusted dynamically. Figure 3-2 shows how the array is created and how it interfaces with the Fixed Rounder block. In the figure, `rsw` is the rounder select width (`rndr_sel_width` in the code), which is the bit width of the `cr_rndr_sel` input. The width of the data input, `d_in`, is controlled by the parameter `input_width`. The width of `q_out` is controlled by `output_width`. When the Fixed Rounder block is instantiated, it uses an input width of `input_width` plus 2^{rsw}-1, and an output width of `output_width`. The radix position for the Fixed Rounder is controlled by `fr_rad_pos` in the Configurable Rounder.

The main purpose of the Configurable Rounder is to take signals and round them at different radix points. This is accomplished by using the sign extension operations and padding least significant bits (lsbs) with zeros to shift the desired radix point to the radix point to which the fixed rounder is set. For example, if the fixed rounder was set to round on the fourth bit, and `d_in` needs to be rounded on the third bit, it can be padded by one zero on the lsb, and sign extended so the width will be the width of the fixed rounder input. The third bit has been shifted

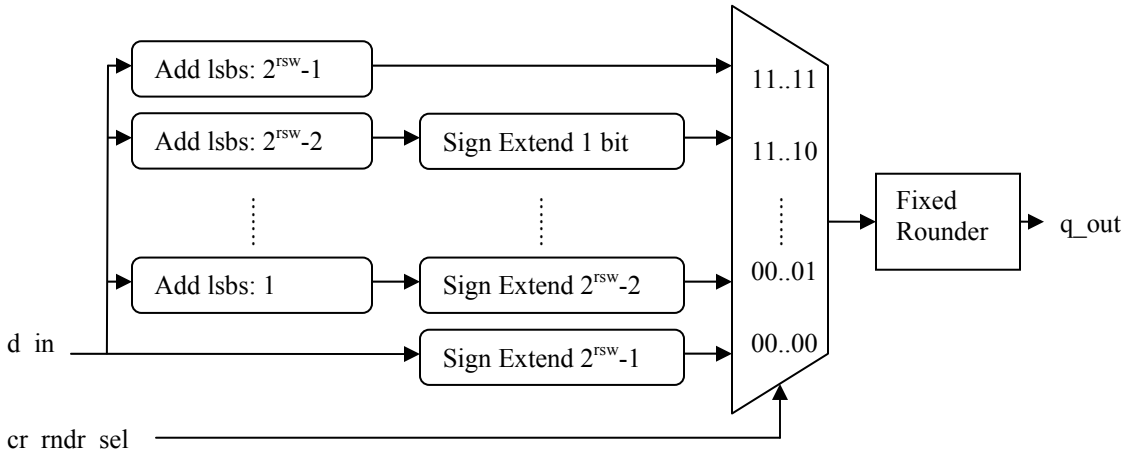


Figure 3-2: Configurable Rounder Structure

to the fourth bit position, so the fixed rounder will round on the correct bit. Another use for the configurable rounder is to shift signals down to the same magnitude. If a series of inputs is known to have different radix points, and the numbers are going to be added together, then they need to be shifted so they all have the same radix point before they are added together. By using the Configurable Rounder, it is possible to select the correct amount of shifting to move the radix point to the desired location. Rounding can be done on the information to decrease the size of the operations that will follow the Configurable Rounder. This algorithm can have problems with making some parameters too large. As the `rsw` parameter increases, the number of inputs to the multiplexer grows proportional to the square of `rsw`.

As with the Fixed Rounder, the Configurable Rounder also has `ri`, `ro`, and `rt` as parameters. In addition to parameters setting the input and output register types and reset type, there is a parameter to set a register on the `cr_rndr_sel` input. This parameter is called `rc`, and it has values associated with it similar to `ri` (0 is no register, 1 is a pipeline register). The `ri` parameter on the Fixed Rounder is set to 0, and the `ro` parameter is mapped to the `ro` parameter of the Configurable Rounder. It should be noted that there are no generate statements in the Configurable Rounder that are based on `ro`, the Fixed Rounder handles `ro` exclusively.

3.1.3 Output Gain Stage

This block is the final type of rounder block. It is the highest block in the rounder hierarchy, as show in Figure 3-3. Also shown are two other blocks of code, `plr.vhd` and `zreg.vhd`. These blocks are the pipeline register and z register, and have their input width and reset type parameterized. This allows them to be used in a variety of situations.

The Output Gain Stage is designed to take an input vector (`d_in`), run it through a Configurable Rounder to make the vector smaller, and then multiply it by another input, `gain`. The output of the multiplier is sent into a Fixed Rounder block so the next block in the DSP algorithm can be loaded with the right number of bits. Figure 3-4 shows the overall layout of the Output Gain Stage. The goal of the Output Gain Stage is to scale an input signal to a range and overall width that will produce useful results in the next processing stage. The Configurable Rounder in this design rounds `d_in` to a smaller value, which makes the multiplier smaller. When the output from the configurable rounder is multiplied by `gain`, the answer has a width equal to the `gain_width` plus the `cr_output_width` of the Configurable Rounder.

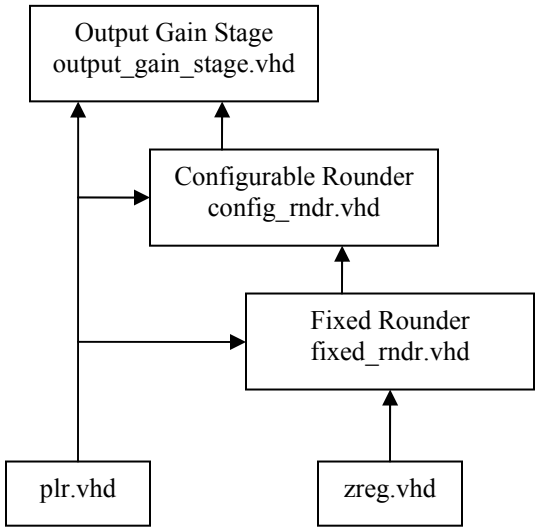


Figure 3-3: Rounder Macro Hierarchy

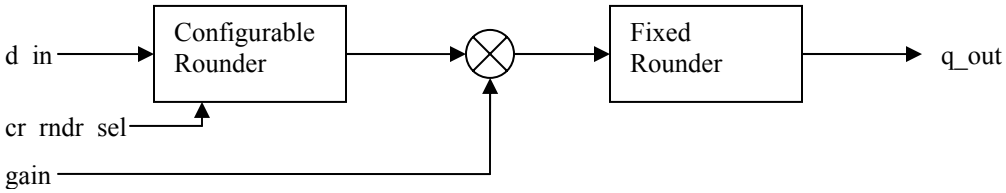


Figure 3-4: Output Gain Stage Structure

By setting the `ri` parameter to one in the instantiation of the Output Gain Stage, a pipeline register is inserted on the input. The input register is handled by the Configurable Rounder block. In the Configurable Rounder block, `ro` is set to zero, which means there is no output register in that block. The `rc` parameter of the Configurable Rounder is inherited from the Output Gain Stage. If `rc` is set to one in the Output Gain Stage, a pipeline register is inserted to delay the value of `gain`. Because the Configurable Rounder inherits `rc`, if there is a register on `gain`, there will be a register on the `cr_rndr_sel` input. Additionally, the `cr_rad_pos`, `rndr_sel_width`, and output width are all set when the Output Gain Stage is instantiated.

An additional register is inserted into the data flow if the parameter `pr` is one. This register is for pipelining the algorithm. When the register is enabled, it is inserted between the Configurable Rounder and the multiplier. No register is inserted between the gain and multiplier. Because of the mismatch in delays, designs using the Output Gain Stage need to take the difference in levels of delay. Another solution to this problem is to not include an extra register on the output for simulation before synthesis, and use a synthesis tool that can allocate pipeline registers into a design automatically.

As in the Configurable Rounder, the Fixed Rounder handles the `ro` parameter of the Output Gain Stage. The `ri` parameter of the Fixed Rounder is set to zero, the input width is set to the output width of the multiplier, and the output width is set to the output width of the Output Gain Stage. The radix position is set when the Output Gain Stage is instantiated.

3.2 FFT

The FFT macro that has been created is using an algorithm called the Radix-2^2 Single Path Delay Feedback (R2^2SDF) algorithm [8]. This algorithm takes a natural order stream of complex number inputs, one per clock cycle, and outputs a stream of complex numbers in bit-reversed order. Bit-reversed order means that if a counter counted each point of data that was coming out, and reversed the order of the bits in the counter, that would be the sample number corresponding to that point of data. For an `N` point FFT, there are `N`-1 registers used in feedback shift registers, 2 $\log_2$ (`N`) complex adders, and ($\log_2$ (`N`) -1)/2 complex multipliers, with any fractions dropped. Log_2 (`N`) is the number of butterfly stages that will be in the design, and are numbered from one to $\log_2$ (`N`). First the components will be described from the lowest level up to the higher levels, then information about the test bench and how the test vectors are created

will be presented. Code for the FFT is in Appendix B. There are also some commercially available FFT cores. For example, Xilinx has one available in their IP Center [9].

3.2.1 Shift Registers, Adders, and Subtract Modules

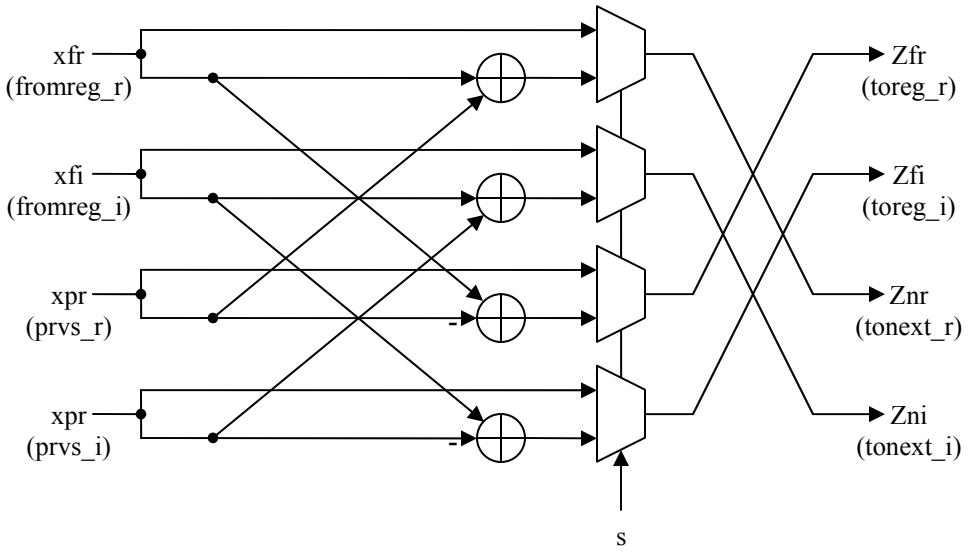
Shift registers are used in this algorithm to provide feedback. Two different modules have been created to fill this role. The first is shiftreg1, which is a 1-stage shift register with the input width set by `data_width`. The second is shiftregN, which is an `N`-stage shift register with input width set by `data_width` as well. In the first stage of the FFT, the shift register is set to `N`/2 stages. Each FFT stage after divides the number of shift register stages by 2, until shiftreg1 is used in the last FFT stage.

Adder and subtract macros are included to reduce the amount of recoding necessary to replace add and subtract operations. It is possible to map operators to macros to different operations. For example, it is possible to map the "+" operator to a macro that adds the two numbers together and returns the sum. If the "+" operator is not mapped to a macro, the adder macro used could be replaced with an adder of a different design. The same applies to normal multiplication and subtraction.

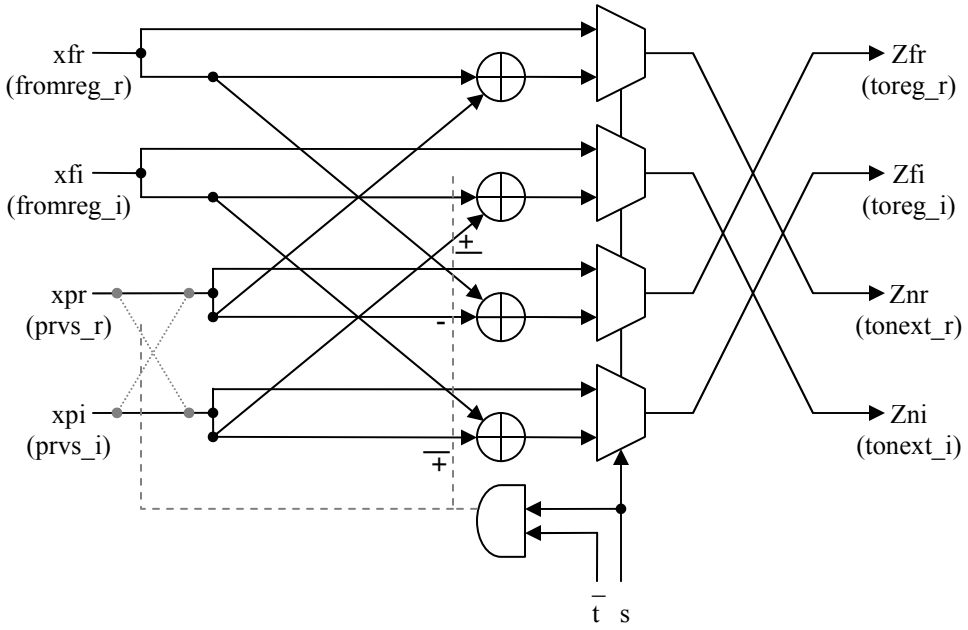
3.2.2 Butterfly Types

There are two different butterfly structures used in this FFT algorithm. The two types of butterflies are shown in Figure 3-5. The names in parenthesis are the names of the signals in the code. The last letter is to signify if it is the imaginary component of a signal or the real component of a signal. As an example, `xfr` and `xfi` are the imaginary and real parts of the complex signal `xf`.

Both butterflies share a control signal `s`, which controls four multiplexers. The signal `s` is driven by a bit in the control counter. The counter has a range of zero to `N`-1. Each stage is driven by a different bit, with the first stage driven by the $\log_2$ (`N`) bit of the counter, and the last stage by the lowest bit of the counter. This signal switches the butterfly between two different modes. When `s` is zero, the butterfly is in passing mode. When in passing mode, `xf` and `xp` are passed to `Zn` and `Zf`, respectively. When `s` is one, the butterfly is in butterfly mode. In this mode, the butterfly operation is performed and the answers sent to the output signals. The BF2I butterfly is used for the odd numbered stages, including the first stage, and the BF2II butterfly is used in the even numbered stages.



(i) Butterfly Type BF2I



(ii) Butterfly Type BF2II

Figure 3-5: Butterfly Structures in Radix2^2SDF Algorithm

The control signal t is used in BF2II to add some extra functionality to the butterfly. When t is zero and s is one, the data in the butterfly is routed differently, and some of the addition and subtraction operations are performed differently. First, the real and imaginary parts of xp are switched. This causes the first and third operations to change from xfr + xpr to xfr + xpi and xfr - xpr to xfr - xpi. The second and fourth operations also change. The second changes from xfi + xpi to xfi - xpr, and the fourth changes from xfi - xpi to xfi + xpr. The code for this is implemented slightly differently. The components of xp are still switched, but instead of changing the type of the second and fourth operation, the outputs are switched. This data manipulation in the BF2II stage is for one purpose. The entire operation mimics multiplying the xp input by $-j$. Performing the operation this way saves space, because a complex multiplier is unnecessary. This multiplication is needed for the algorithm to function correctly, and is done to simplify some of the twiddle factor multiplication. The control signal t is equal to the s bit of the previous stage.

Both butterflies have two parameters, the output_width and add_g parameters. The add_g parameter is the adder growth, and can be either zero or one. Whenever binary numbers are added, the output width grows to be one bit bigger than the input width. The add_g parameter regulates if the extra bit is added onto the signal, or if the signal will remain the same size. For example, if the output width is 12 and add_g is set to one, the input width will be 11. The signal xp is the only input signal set by input_width, all the rest are set by output_width. The signal xf is set by output width because it is input from the feedback shift register. The shift register has Zf as the input, so the width of xf and Zf must match.

Suppose an instantiation of BF2I has an output_width of 13 and an add_g of 1. In order for the adders to function properly in this version, they must have the same width input and output. The signal xp must be sign extended to match the width of xf. The sum of the different components will then be 14 bits. A special multiplexer, mux2mmw, is inserted to handle the signals with mismatched width. The output from this multiplexer is 13 bits wide.

The highest bit in the adder can be truncated because of how the butterfly and shift register work together. The butterfly in each stage is controlled by a bit in a counter. The controlling counter will be discussed more in a later section. Because it is a counter, the s signal will only be high for a number of clock cycles equal to the depth of the shift register for that stage. For example, the first stage in a 64-point FFT has a shift register that is 32 stages deep.

For the first 32 clock cycles, the butterfly is in passing mode, and is filling up the shift register with the 12-bit inputs. For the next 32 clock cycles, the butterfly is active, and the shift register will be filling inputting the 13-bit results of the butterfly operations, and will be outputting the stored 12-bit values. For the next 32 clock cycles, `s` is low again, so the shift register will be outputting the stored 13-bit values, and storing new 12-bit values. The process then repeats itself. Never is a 13-bit number added to a 12-bit number, only 12-but numbers are added to 12 bit numbers. The same is true of the BF2II stage.

3.2.3 Complex Multiplier

At each even-numbered stage, except the last if it is even, the `Zn` complex signal from the BF2II butterfly is multiplied by a complex twiddle factor. Figure 3-6 shows the structure of the complex multiplier. The input width of the two inputs can be different. The addition and subtraction operations are performed on signals of the same width, since every input to those operations comes from the product of one part from each signal. This means that real and imaginary parts of each signal must be the same width.

The multiplier growth parameter (`mult_g`) is used to limit the output of the multiplier. In this design, each instantiation of the complex multiplier is multiplying by the same width signal for the second input, called twiddle factor width. The first input is the output from the BF2II stage, and the second is the twiddle factor, from the ROM storing the twiddle factors. The output from the multiplier is equal to the output width of the BF2II output plus the parameter for

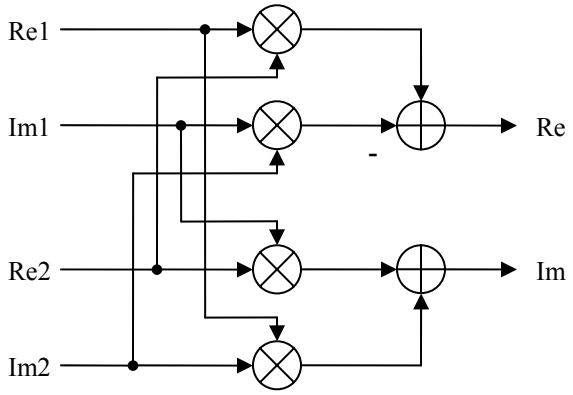


Figure 3-6: Complex Multiplier Structure

the twiddle factor width, `twiddle_width`, plus one. The additional one comes from growth during the addition and subtraction operations. In the complex multiplier, `add_g` is part of `mult_g`. The output from the complex multiplier is truncated to be equal to the first input's width plus the `mult_g` parameter. This limits the `mult_g` parameter to a range from zero to the `twiddle_width` plus one.

3.2.4 Twiddle Factors

In the R2^{2}SDF algorithm, every even stage has a complex multiply by a complex twiddle factor. In this version of the algorithm, the twiddle factors are generated ahead of time in MATLAB and written to a VHDL file that contains a case structure. The case structure it generates is similar to a ROM, and a ROM could be substituted for it easily. It will be referred to as the twiddle factor ROM or just ROM. The MATLAB code to generate the ROMs is in Appendix C, as is all the other MATLAB code.

The following equation is used to generate the twiddle factors. In this equation, `N` is the

$$W_i(x) = e^{(-2j\pi x(k/N))} \tag{2}$$

total number of points in the FFT, and `x` ranges from zero to the number of points in the ROM divided by 4. The number of points in the ROM, `m` is equal to the number of shift registers in this stage multiplied by four. The variable `k` changes as a function of the number of points. For the first `m`/4 points, `k` is equal to 2*`N`/`m`. For the second `m`/4 points, `k` is equal to `N`/`m`. For the third, k is equal to 3*`N`/`m`. For the last `m`/4 points, `k` is equal to zero. The result of taking the e^0 is one for the real part and zero for the imaginary part, so the last `m`/4 points are one for the real parts and zero for the imaginary.

The width of the twiddle factor is controlled by the `twiddle_width` parameter. When MATLAB generates the twiddle factors, it is initially not in a fixed-point format. To scale them, they are multiplied by $2^{\texttt{twiddle\_width}}$-1. Any fractions are dropped, and then they are converted to two's compliment form. This is done separately on the real and imaginary parts of the twiddle factors.

To make the process of generating the ROMs easier, a different MATLAB program is designed to repeatedly call the ROM generator program. It generates all the ROMs needed for a specific `twiddle_width` and number of points.

3.2.5 Control Logic

The control logic for the FFT is simple. A counter is used to switch the butterflies between modes and pick the twiddle factors from the ROMs they are stored in. The counter is $\log_2$ (`N`)-bits wide. In addition to the `clock` and `reset` signals, it uses a `load_enable` signal to start. In addition to controlling when the counter starts, it controls how long the FFT runs. The counter is designed to continue counting if the load enable is set to low and the load enable signal was high for at least `N` clock cycles. This allows the FFT to keep outputting data for which it has the entire waveform. If the load enable is set to low after less than `N` clock cycles, there will be an error and the FFT will need to be reset.

The FFT takes in a stream of complex numbers. The first clock cycle when the load enable is high is when the first data point is read. On the `N`-1 clock cycle, the counter is equal to its maximum value, and the output from the FFT is dependent on the input. It takes the output on that clock cycle longer to stabilize than any other clock cycle.

3.2.6 Structure

The structure of the FFT is built based on the parameters. The number of points (`N`) sets how many stages the system will have (`num_stages`) to $\log_2$ (`N`). The bit width of the input is controlled by the `input_width`, and how many bits wide the output signal will be is set by the number of stages, the multiplier growth, and the adder growth. Figure 3-7 shows a flowchart of how the structure is generated.

After initializing a counter to one, a generate loop runs from `i`=1 until `i=num_stages`. During each iteration of the loop, several conditional **generates** assign the correct butterfly for that stage and connect the inputs and outputs of the butterfly to the correct signals. The first stage has its inputs connected to the input signal of the FFT. When each stage is generated, it has an input width equal to the following equation. The division of `i` minus one by two, when

$$stage\_width = input\_width + add\_g * i + mult\_g * ((i-1)/2) \quad (3)$$

performed in VHDL, does not return the fractional part of the answer. The equation generates the bit width of the output of the butterfly in each stage. Subtracting `add_g` from the equation gives the input width. In the BF2I butterfly, the `stage_width` is equal to the output width. In the BF2II butterfly, the output width is equal to `stage_width` plus the multiplier growth parameter.

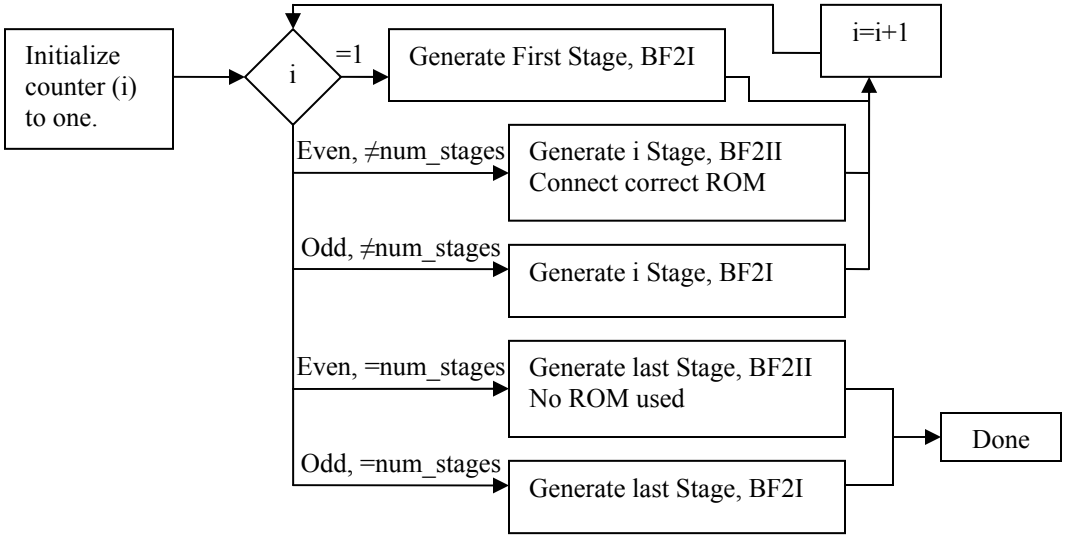
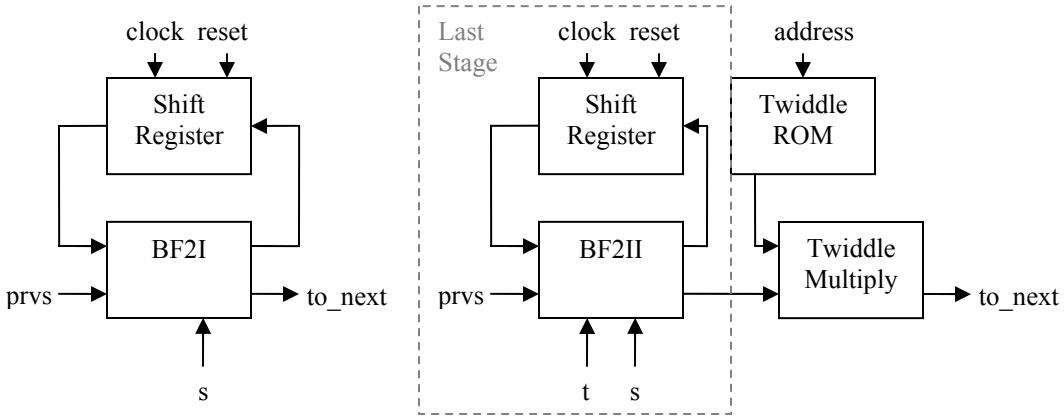


Figure 3-7: Top Level FFT Generation Flow

In order to hold the interconnections, an array of signals is created. The width of the signals in the array is set by the equation above, substituting `num_stages` in for `i`. The number of signals in the array is equal to `num_stages` minus one. Only the parts of the signal that are used will be synthesized. For example, if the output width of a stage is 13 bits and the output bit-width is 36 bits, 23 bits of the array entry that correspond to that stage output are not connected to any signals. Since those signals are not connected to anything, they will not be synthesized. In Pre-synthesis simulation they can still be seen, however

The output from the last stage is connected to the output of the FFT macro. If the last stage is an odd-numbered stage, then the BF2I butterfly is used. If the last stage is an even-numbered stage, the BF2II butterfly is used, but there is no twiddle factor multiplication. Figure 3-8 shows what the structure of each stage looks like. Except for the control signals (`clock`, `reset`, `address`, `t`, and `s`), all the signals in the stages are complex. The gray box labeled "Last Stage" in Figure 3-8(ii) shows which components of stage_II are used in the last stage if it is an even-numbered stage.

The shift registers in Figure 3-8 are actually two shift registers in parallel. One is for the real part of the signal; the other is for the imaginary part. The width of each shift register is `stage_width` bits, and the depth is $2^{\texttt{num\_stages-i}}$. So for the first stage of a 64-point FFT, the shift register is 32 stages deep. The address line controlling the twiddle factor ROM is



(i) BF2I stage: stage_I (ii) BF2II stage: stage_II

Figure 3-8: FFT Stage Structures

`num_stages` minus `i` plus two bits wide. The address line width is equivalent to taking the counter output from the bit tied to `t` down to zero. So, for the fourth stage in a 64-point FFT, the twiddle factor ROM has 16 entries and the address signal is connected to the lower four bits of the counter (from the `t` bit down to the lowest bit). The Twiddle Multiply block is a wrapper for the complex multiplier. It contains the complex multiplier and truncates the output of the multiplier to a width equal to the width of the butterfly output plus the multiplier growth.

Figure 3-9 is an example of a 64-point FFT. The input width is set to 12 bits, the twiddle width is set to 10, the multiplier growth is set to 9, and the adder growth is set to 1. As with Figure 3-8, all the signals in Figure 3-9 are complex except the control signals. The widths of the signals are labeled, and the widths are for each part of the signal (real and imaginary).

Rather than clutter the figure with the control lines, the output from the counter is labeled. The most significant bit in the counter is labeled `c5`, and the number decreases to the least significant bit, `c0`. The control signal `s` for the first butterfly is tied to `c5`. The formula for determining which control bit a stage uses is `num_stages-i`. The address line of the twiddle factor ROM for the second stage is labeled `[c5..c0]`. This notation means the most significant bit in the address is tied to the highest bit in the counter, and second most significant bit is tied to the second highest bit, and so on down to the least significant bits of each signal being connected.

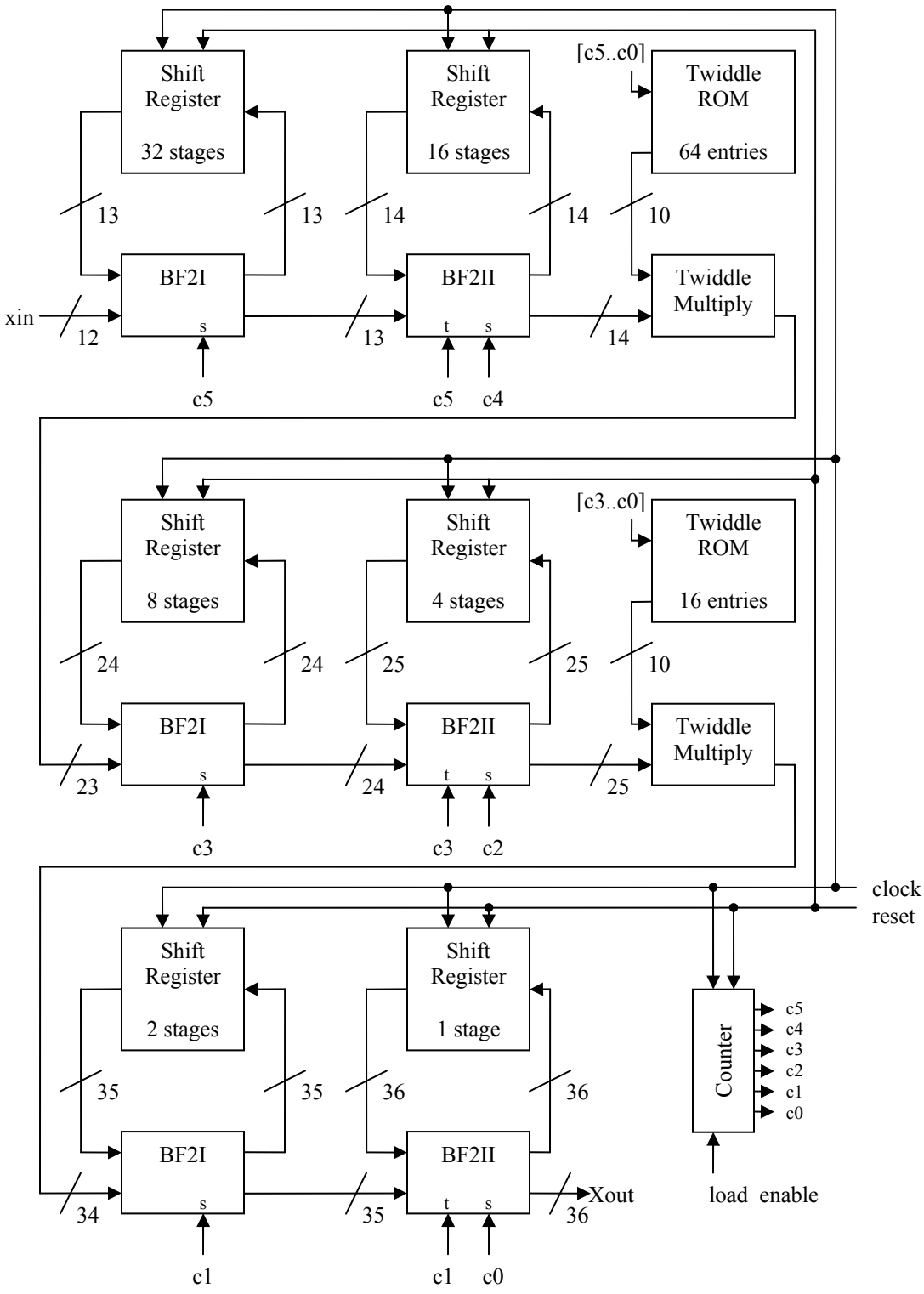


Figure 3-9: Generated Structure of 64-point FFT

As was mentioned earlier, on the clock cycle when the last point of data is on the input has the longest delay. On that clock cycle, all the butterflies are in active mode and the complex multipliers are operating along the path. By tracing the data path through Figure 3-8, the amount and type of operations can be seen. The data must travel through six addition or subtraction operations and two complex multiply operations. As the number of points grows, the number of multipliers and additions/subtractions increases, which increases the delay. For a 1024-point FFT, the data will travel through ten addition/subtraction operations and four complex multiply operations. Increasing the bit-width of the input or the twiddle factor will also increase the delay, because the arithmetic operations take longer to reach an answer for larger inputs. Controlling the growth of the signal is the purpose of the `mult_g` parameter. By decreasing the `mult_g` parameter, the growth of the signals will be smaller, which will translate to less delay. However, truncating the output of the multiplier will decrease the accuracy of the results.

The VHDL files included are configured to match the parameters in this example. The hierarchy for the VHDL macros is functionally equivalent to what is presented here. The generate loop shown in Figure 3-7 is in the top-level module `fft_core.vhd`. This loop connects the different types of stages (`stage_I.vhd`, `stage_II.vhd`, `stage_I_last.vhd`, and `stage_II_last.vhd`) together. The only difference between `stage_I.vhd` and `stage_I_last.vhd` is that `stage_I.vhd` uses `shiftregN.vhd` to instantiate the shift registers, and `stage_I_last.vhd` uses `shiftreg1.vhd` for its 1-stage shift registers. The even-numbered stages that are not the last stage are implemented using `stage_II.vhd`. This module contains the BF2II butterfly (`BF2II.vhd`), two copies of the shiftregN macro, and the twiddle multiplier (`twiddle_mult.vhd`). It has inputs that are wired to the twiddle factor ROM for that stage. The ROMs are numbered from largest to smallest, so the first ROM in the data path is always `rom1.vhd`. The other type of stage II module, `stage_II_last.vhd`, is identical to the `stage_II.vhd` module, except that it does not include the complex multiplier or the inputs from the ROM.

3.2.7 Test Bench

The test bench for the FFT is written in VHDL, but it loads test vectors that are created in MATLAB. The vectors control the FFT and send three input waveforms into it. The second vector is repeated as the third, and the only purpose for it is to enable the test bench to keep writing the output of the second waveform until it finishes. After the clock is started, the test

bench writes the output of the FFT to a file. By default, it reads from a file called "testvec" and writes to a file called "data.out".

The test vectors generated are a concatenation of the `reset`, `load_enable`, `xin_r` (the real part of the input), and `xin_i` (the imaginary part of the input). In the files the bits of the signals are ASCII characters, not stored binary numbers. This makes it easy for a user to examine the files and verify that they have been generated correctly. By putting the `reset` and `load_enable` into the test vectors, the only signal that needs to be set in the test bench is the `clock`. However, having the `reset` operating at almost the same time as the clock can cause problems in simulations after synthesis. In this case, it is better to force the `reset` signal high during the clock cycle before the `reset` would be set high by the test bench.

The advantage of writing the vectors in MATLAB is that creating an approximate model of the FFT is easy. After the test bench has been simulated, a second MATLAB file, `plotdata.m`, can read the original test vectors, convert them to floating point numbers, perform MATLAB's built in FFT function on them, and finally plot them and the results of the FFT simulation. This allows for easy visual comparison of the outputs. It can be modified to display data in different ways (real and imaginary or magnitude and phase), depending on what test vectors are input. If the Pre-synthesis results are correct, a new test bench can be created to read the Pre-synthesis results and compare them to the FFT being simulated.

In addition to a MATLAB test bench, a BIST module for the 64-point FFT has been created. This module operates as was described in section 2.4.3. The purpose of this module is to serve as an inexpensive alternative to external testing hardware. It is not reconfigurable at this time, but the components except for the ROM are.

Chapter 4: Results

4.1 Rounders

The test bench for each rounder module instantiates the macro under test, sets up an internal clock, and then inputs the test vectors into the macro. Each macro had over 1000 test vectors. The `q_out` signal is compared to the `expected_q_out` signal each clock cycle, and if they were not equal, an error is reported. The parameters in each simulation are displayed as integers, the control signals (if any) as binary numbers, and the data signals as hexadecimal numbers. When referring to binary numbers, 0b will be before the number. Hexadecimal numbers in text will have 0x in front of them. Due to the length of the rounder test benches, they are not included in the Appendices.

4.1.1 Fixed Rounder

Figure 4-1 shows a sample of the Fixed Rounder simulation. The figure shows five test vector inputs and outputs. This is a Pre-synthesis simulation, so the outputs appear to be changing at the same time as the input. This is an effect of using a Δ delay in an event driven simulation. In this simulation, there are no input or output registers (`ri` and `ro` are equal to zero); the input `d_in` is a 23-bit wide signal. The rounding is set to be on the 10^{th} bit, and the final output width is set to 12 bits. This means the output from the rounding section (`rounded`) will be clipped down to 12 bits from 13.

The last input shown, 0x5652C7, is an example of a negative number that is too large to fit in a 12-bit number, so the output is set to the maximum 12-bit signed number, 0x801. The second and fourth test vectors are both positive numbers that are too bit to fit in a 12-bit number, so the output is set to 0x7FF, the maximum positive output. The 10^{th} bit of the first test vector is one, so that test vector shows a successful positive round. The 10^{th} bit of the third test vector is a zero, so that vector shows a case where a positive number is not rounded.

Initially, one test case was missed in the test vectors. In the clipping section, it was possible for a negative number to be set to outside the intended range of outputs. For example, if the input was 0b111000, the rounding position was set to zero, and the output width set to 5, the Fixed Rounder would return 0b1000, instead of the intended 0b1001. The code was modified to account for this case, the test vector added to the test bench, and the Fixed Rounder was successfully simulated for all the test vectors.

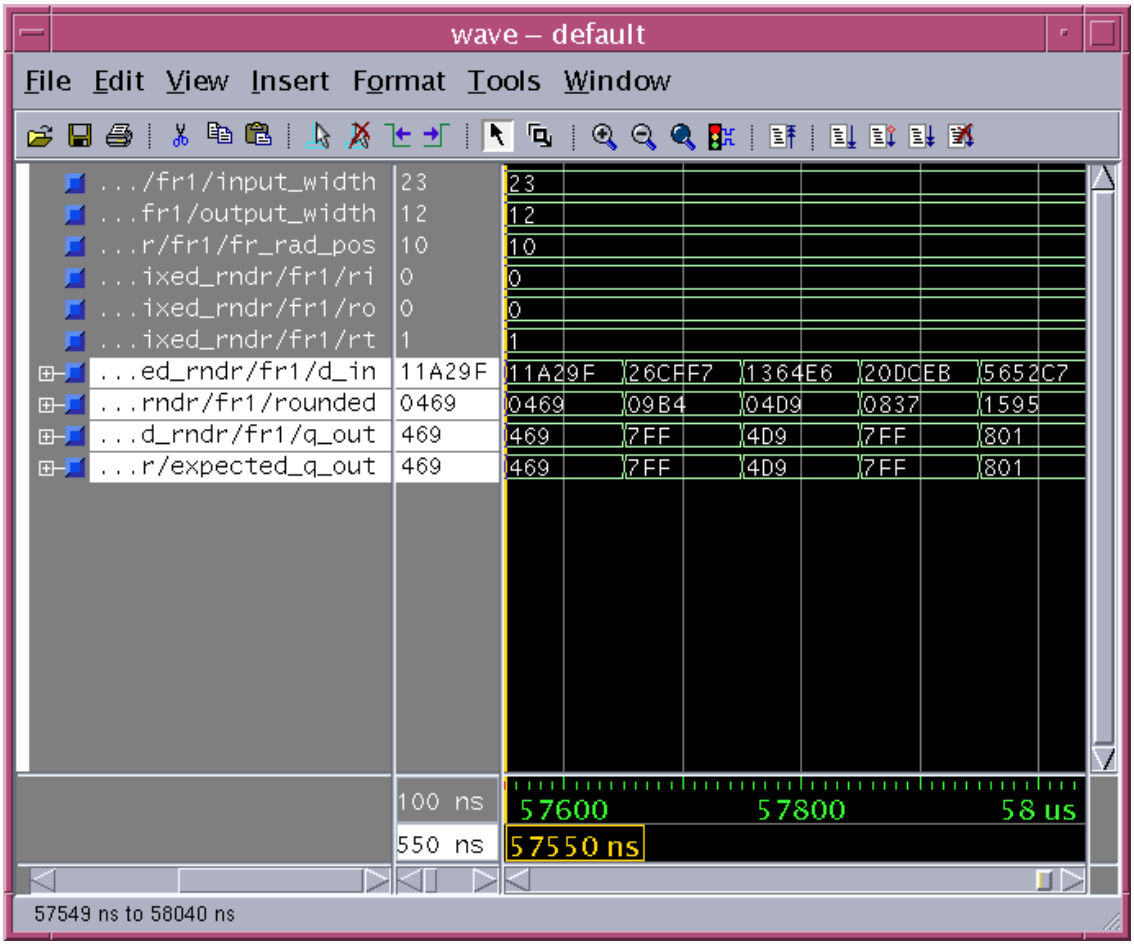


Figure 4-1: Sample of Fixed Rounder Test Bench Simulation

4.1.2 Configurable Rounder

Figure 4-2 shows a sample of the test bench simulation for the Configurable Rounder. The test bench sets the input width to 23, the output width to 12, the rounder select width to 4, and the radix position (`cr_rad_pos`) to 11. It also instantiates a pipeline register on the output by setting `ro` to one. The pipeline register is set to a synchronous reset by setting `rt` to one. The `out_load_enable` signal shown is not used, because the output register is not set to two. Since there is a register on the output, the q_out shown is the rounded output for the previous input. It should be noted that the Fixed Rounder block used in this simulation has different parameters than the Fixed Rounder in the previous simulation.

The `ext_d_in` signal shown is the array that is created in the configurable rounder. In the figure, it is broken down into its four component signals, labeled `(0)` to `(3)`. Entry `(0)` corresponds to the signal in Figure 3-2 that is just sign extended, and `(3)` corresponds to the signal that is just padded.

The Configurable Rounder simulated correctly for all input vectors.

4.1.3 Output Gain Stage

Figure 4-3 shows part of the test bench simulation for the Output Gain Stage. The parameters controlling the width are displayed in the simulation. Because of the number of parameters in this design, the parameters controlling the insertion of registers were left off the simulation output. In this test bench `ri`, `ro`, `rc`, `pr`, and `rt` are all set to one. This will insert a pipeline register with a synchronous reset on the inputs `d_in`, `cr_rndr_sel`, and `gain`. A pipeline register will be inserted between the Configurable Rounder instantiation and the multiplier, and on the output of the fixed rounder. This leads to a complicated data flow. For example, the fifth `q_out` in Figure 4-3 is 0x0F6. The `d_in` and `cr_rndr_sel` used to calculate this answer are 0x0004EE0 and 0b011, which are input to the system three clock cycles before the output is ready. The `gain` used, 0x32, is input two clock cycles before the output is ready.

The Configurable Rounder block used in this simulation has different parameters than the macro used in the previous simulation. The Fixed Rounder inside the Configurable Rounder has different parameters than the two used in previous simulations. The second Fixed Rounder in this simulation has the same parameters as the first Fixed Rounder simulation, except `ro` is set to one.

The Output Gain Stage simulated correctly for all test vectors.

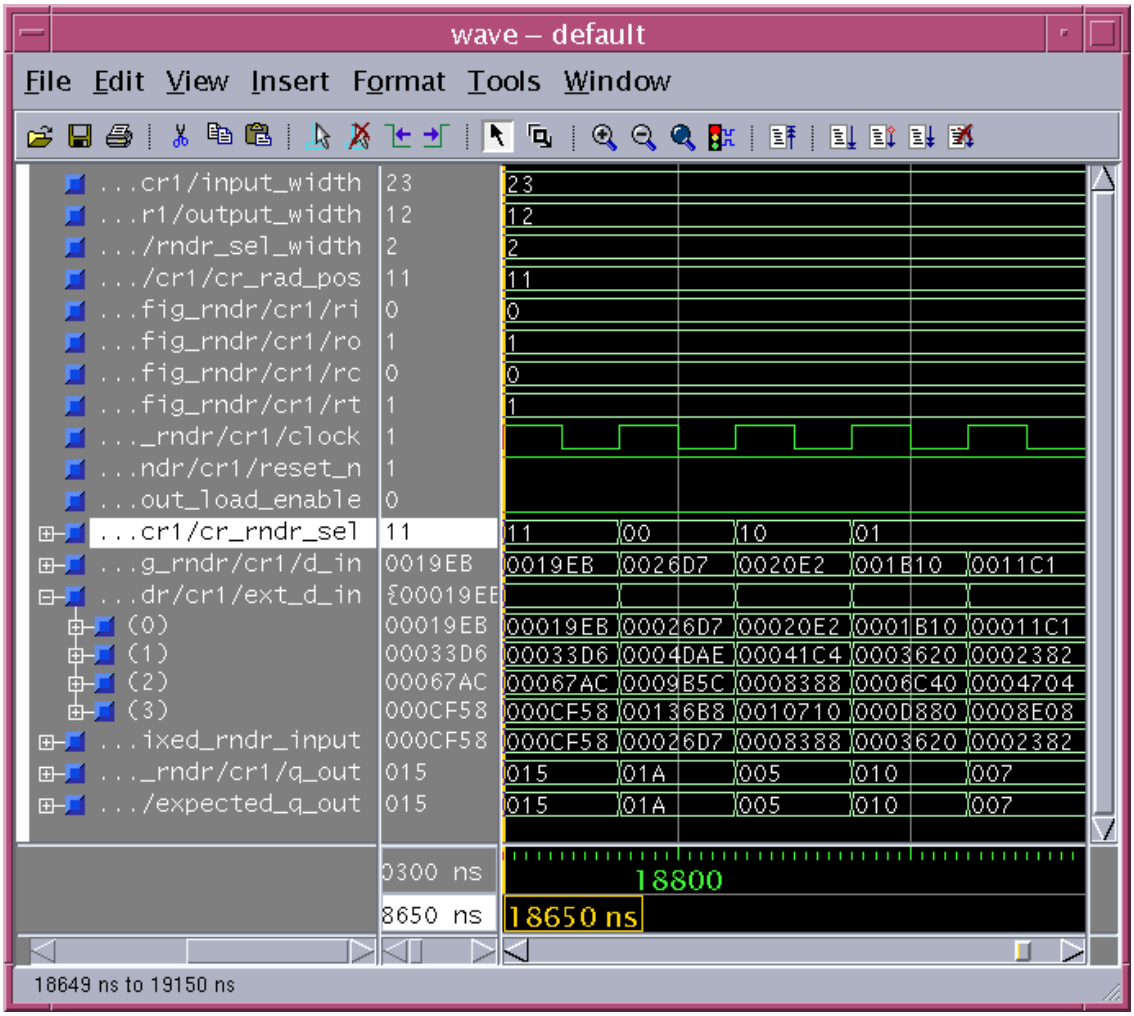


Figure 4-2: Sample of Configurable Rounder Test Bench Simulation

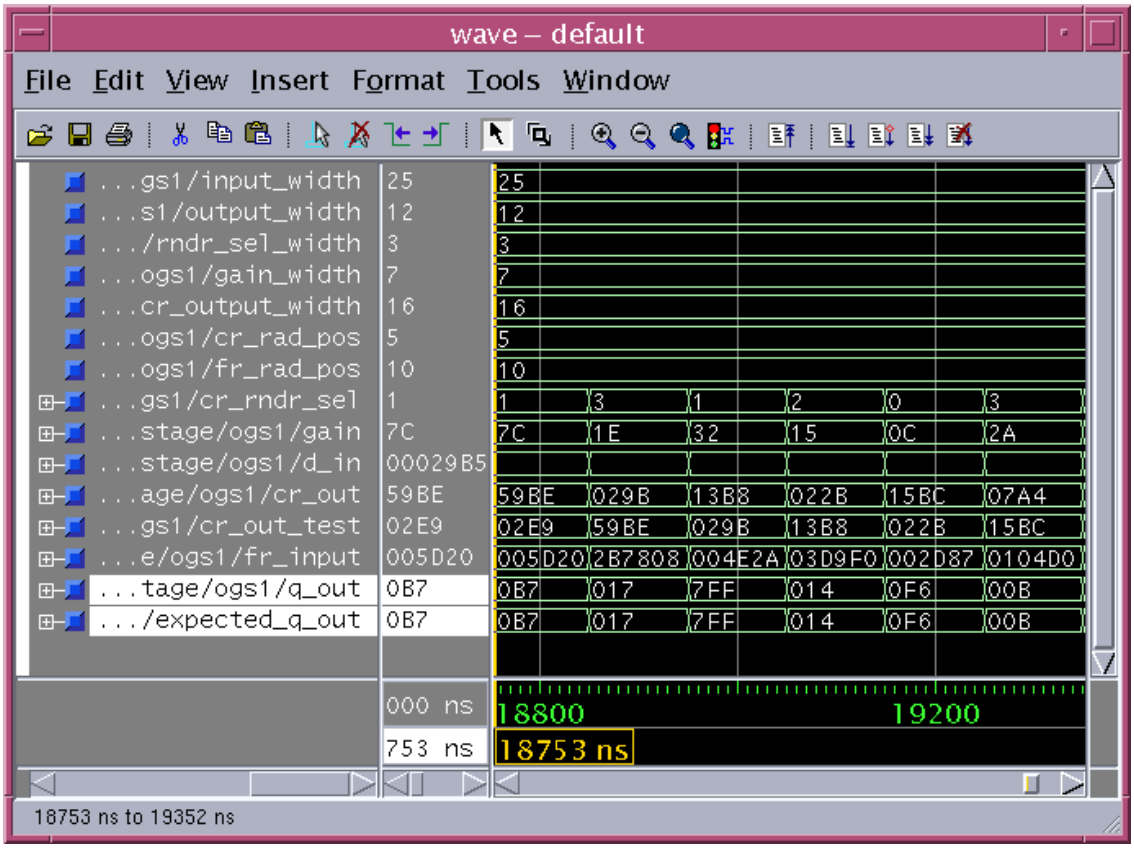


Figure 4-3: Sample of Output Gain Stage Test Bench Simulation

4.2 FFT Results

The simulations in this section were run using the test bench described in section 3.2.7. MATLAB was used to create the test vectors, and the same test vectors are used in all the simulations where `N` is equal to 64. Except for the section covering FFT results from different numbers of inputs, all the simulations are using a 64-point FFT.

4.2.1 MATLAB

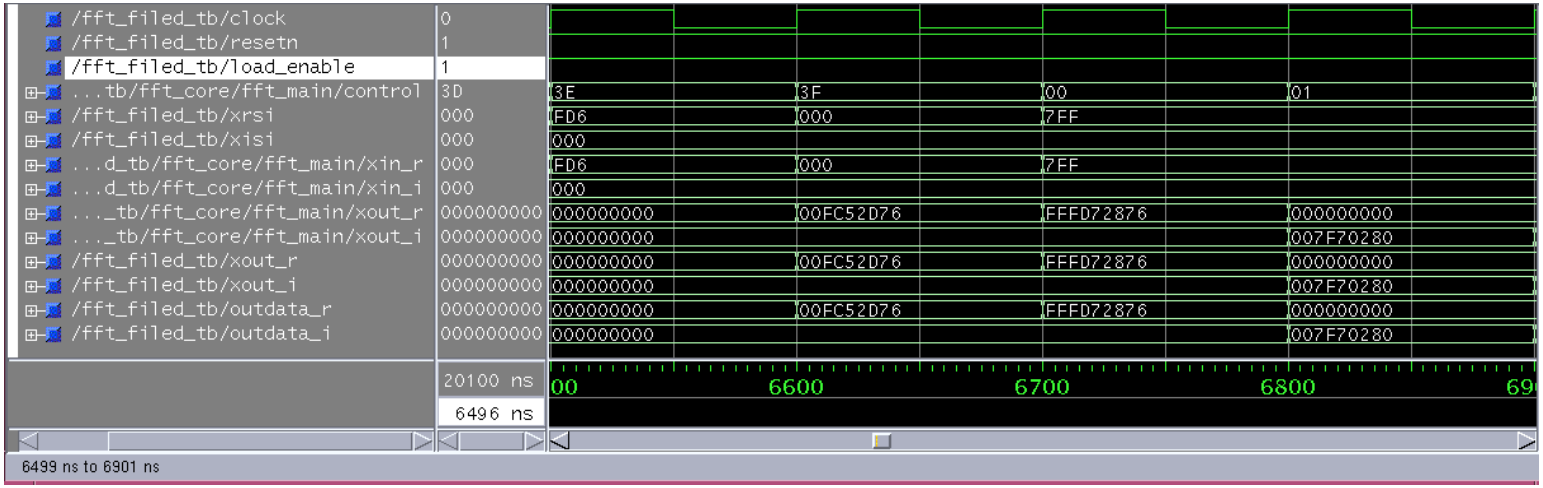
MATLAB is used to generate the test vectors and to compare the outputs to what MATLAB calculates them to be. The test vectors that are generated by default are a sinc wave followed by a square wave. These are used because in the frequency domain they change from one to another. So the FFT of the sinc wave is a square wave, and the FFT of a square wave is a sync wave.

All the simulations in the following sections use the same formulas to determine the test vectors. The Sinc Wave input uses sinc((x-N/2)/2), and the square wave has a duty cycle of 0.125.

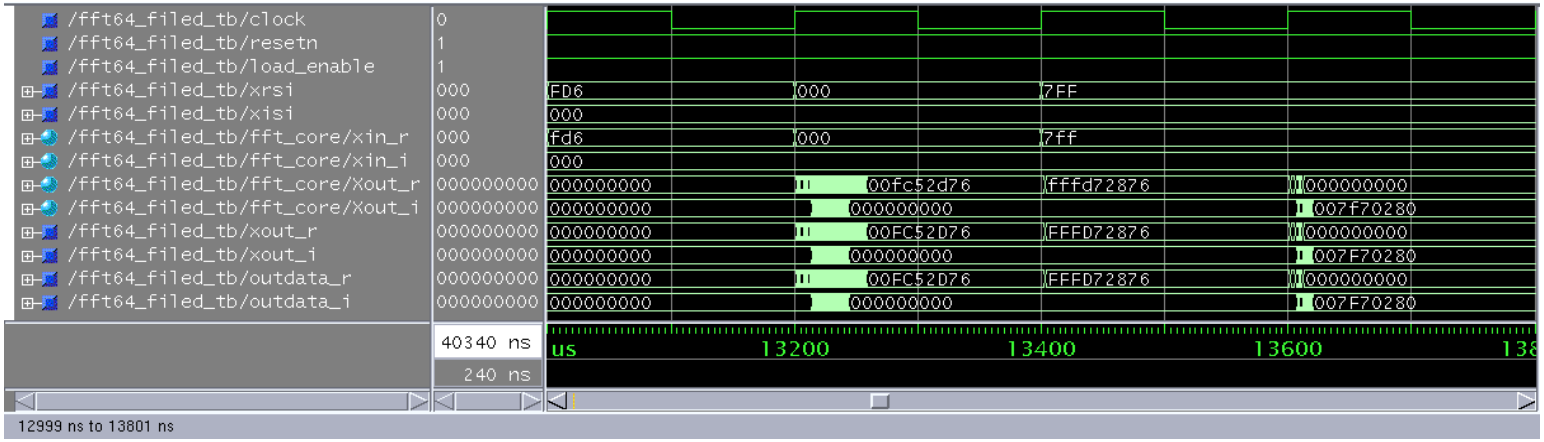
4.2.2 Pre-synthesis and Pre-layout

Figure 4-4 shows a comparison of the same clock cycles for the Pre-synthesis simulation and the pre-layout simulation. Figure 4-4(i) shows the Pre-synthesis simulation. This simulation is event driven and is using a Δ delay. Figure 4-4(ii) shows the pre-layout simulation, and is using a fixed delay of 1 ns per gate and is also event driven. Due to some timing issues with how the test bench writes the data out (it writes to the file on the falling edge), the pre-layout simulation is using a clock with a 200 ns period, while the Pre-synthesis is using a 100 ns clock period. The figures are showing the waveforms near the clock cycle at the end of the first input vector. The `control` signal in Figure 4-4(i) shows the value of the counter in the main FFT.

When control is 0x3F (the maximum value for the counter), the last vector from the sinc wave input is shown in xrsi (real component) and xisi (imaginary component). The Pre-synthesis simulation, since it has no real delay for the logic, produces the output faster than the pre-layout simulation. The lines on the output signals that go from 13200 ns until the value of the signal is displayed are the transitions the outputs are going through. The signals eventually stabilize to the correct values. When the test bench was running with a 100 ns clock, the data being written to the file was incorrect, because the signals had not stabilized yet. After the clock was changed to a 200 ns period, the pre-layout simulation operated correctly.



(i): Pre-synthesis Simulation



(ii): Pre-layout Simulation

Figure 4-4: Pre-synthesis and Pre-layout Simulation Sample

On the third clock cycle, the delay in the post-layout simulation is much smaller. This is because the `control` signal is now all zeros, so all the butterflies are in passing mode. When the butterfly in the last stage is in passing mode, the output of the FFT will be the value that was stored in the 1-stage shift register. Every clock cycle where the lowest bit of the `control` signal is zero, the output from the FFT has a short delay.

4.2.3 Layout and Post-layout

The layout for the 64-point FFT was placed and routed using a TSMC 0.18 micron process, using standard cells for the logic functions. This process has 6 metal layers, which are the most visible element of the layout in Figure 4-5. In order to leave space for routing over the circuit, only the lower 4 metal layers were used. After PAR, the circuit delays were back-annotated and the simulation checked again to see if the layout would simulate correctly. The layout simulated the test vector correctly. Figure 4-6 shows the results of the post-layout simulation.

Overall, the layout is 610.5 μm by 610.4 μm. The total area is 372649.2 μm^2. The layout has 159,074 transistors for the 64-point FFT. For the 256-point FFT, the number of transistors grew to 455,305. For the 1024-point FFT, with `mult_g` set to 4 instead of 9, the number of transistors grew to 1,268,238.

4.2.4 Hardware Testing

The FFT macro was fabricated with a BIST, due to the large number of inputs and outputs. The BIST was designed to be the only operating mode of the chip, but otherwise was designed as shown in Figure 2.3. On a reset, the BIST would start. The load enable signal in the FFT is controlled by the BIST. On the first clock cycle after a reset, all zeros are input to the FFT. On the second, the load enable signal is set to high, and the first test vector is loaded. The test vectors in the ROM are the same vectors that were generated by MATLAB. The signature was generated in a post-layout simulation of the BIST. The signature compressor takes each part of the output, real and imaginary, and uses two 36-bit LFSR structures to generate two 36 bit signatures, which it compares to the signatures that were stored.

Figure 4-7 shows the layout of the chip that was fabricated. There were several groups involved in this project, so there are some parts of the chip that were not tested. Each FFT module was tested first at 10 Hz, to make sure there were no timing problems with generating the output. Then, they were tested at 1 kHz and 1 MHz, to verify the designs were working. The

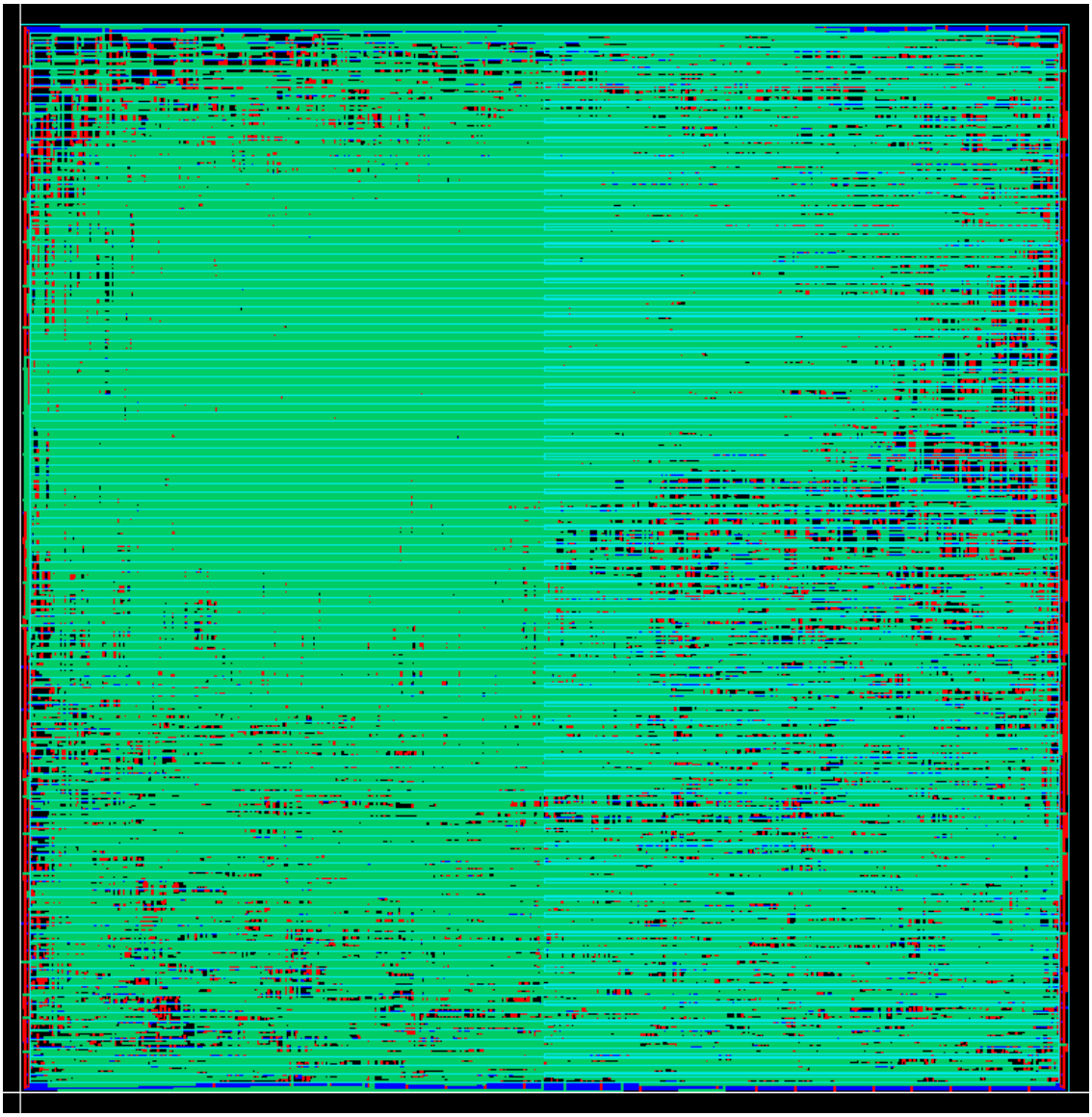


Figure 4-5: Layout of the 64-point FFT

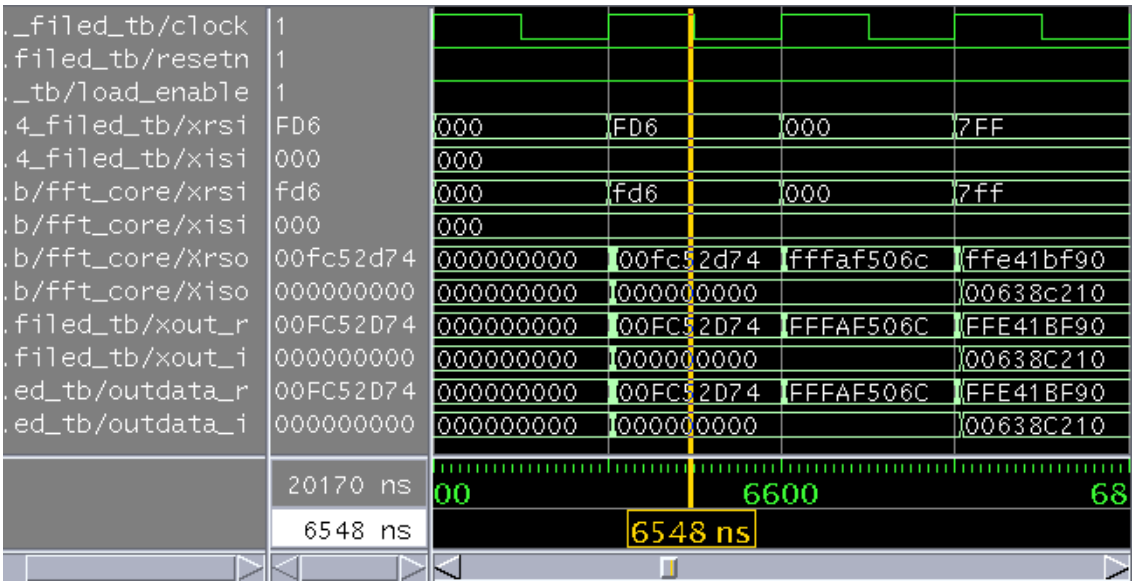


Figure 4-6: Post-layout Simulation Sample

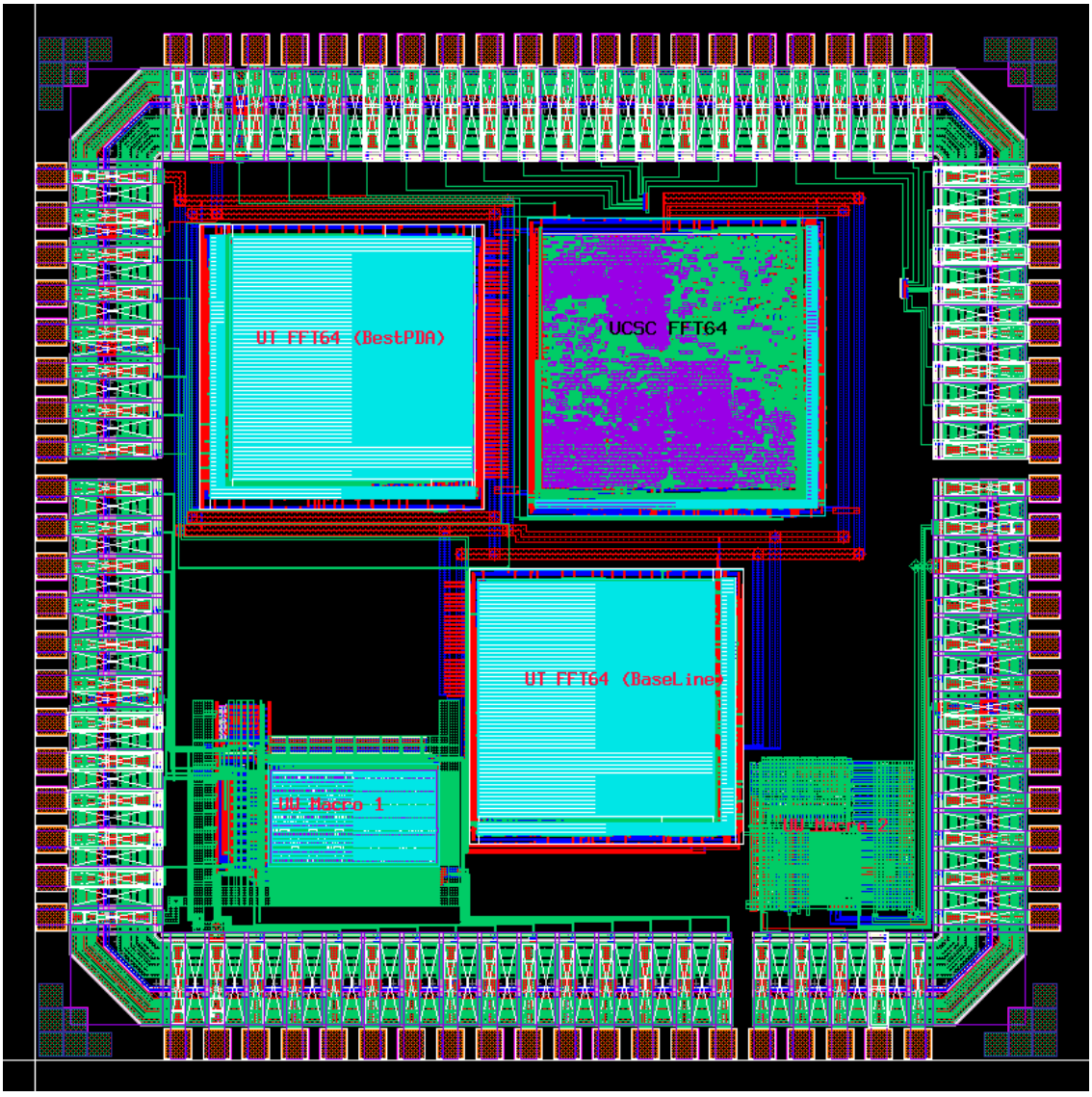


Figure 4-7: Layout of Fabricated Chip

baseline module and the best PDA (power-delay-area product) module implemented at the University of Tennessee worked correctly. The FFT module implemented by the University of California at Santa Cruz is still being tested at this time. Further testing needs to be done to accurately measure the power used by each module and to find the maximum frequency that the modules operate at.

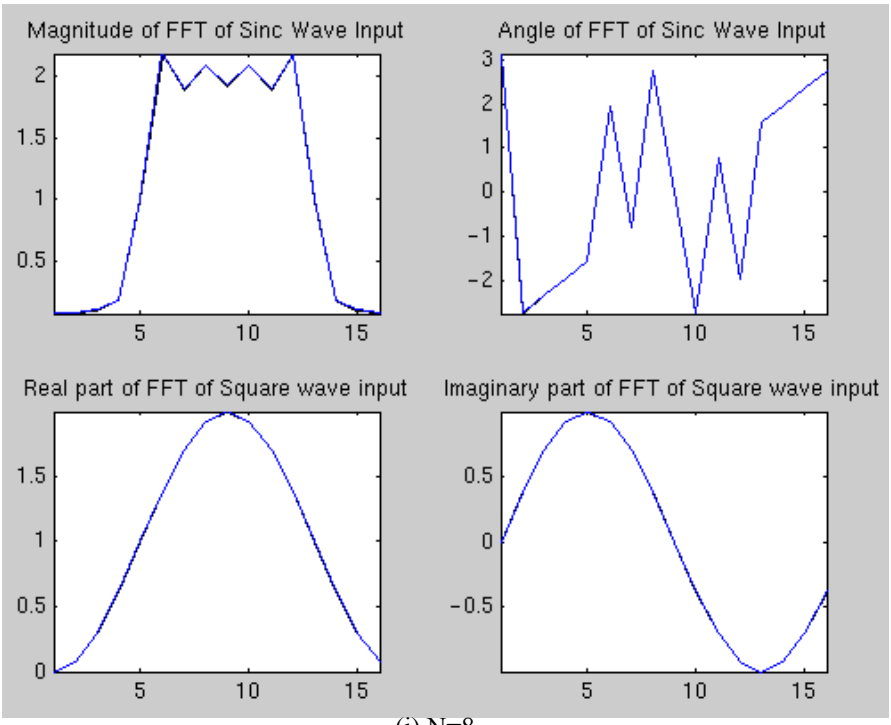
4.2.5 FFT Flexibility

Figure 4-8, 4-9, and 4-10 show the output of Pre-synthesis simulation of the FFT using different numbers of points. The other parameters remain the same, except for the 1024 point uses a `mult_g` of 4 instead of 9, to cut down on the amount of logic used. Figure 4-8(i) shows an 8-point FFT simulation. As the number of points increases, the FFT of the sinc wave input becomes more like an ideal square wave. The increasing noise in the angle of the square wave output is from the smaller values at each point. The inaccuracy of a digital representation is significant compared to the values of the signals at those points, which manifests in the angle more than in the magnitude.

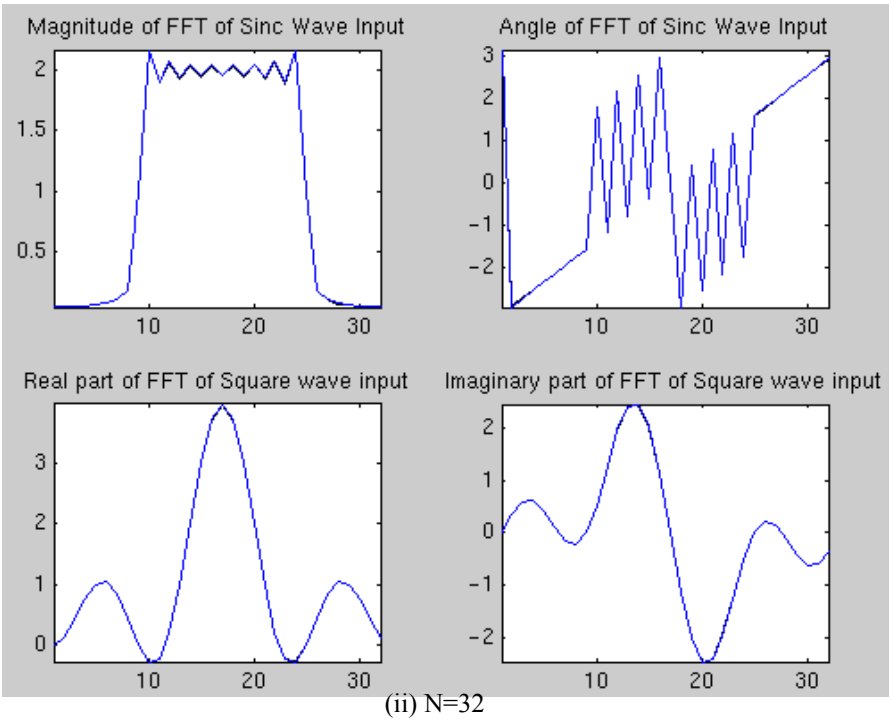
The FFT was also synthesized to 250 and 130 nm processes. Table 1 shows the estimated PDA, normalized to the 180 nm process. The operating voltages are also shown. The power was estimated at a 20 MHz operating frequency in the pre-layout testing stage. The table shows 8-fold improvements between processes.

Table 1: Normalized PDA Estimates for Different Processes

Feature Size	250 nm	180 nm	130 nm
Operating Voltage	2.5v	1.8v	1.2v
PDA Normalized to 180 nm	0.15	1.00	8.74

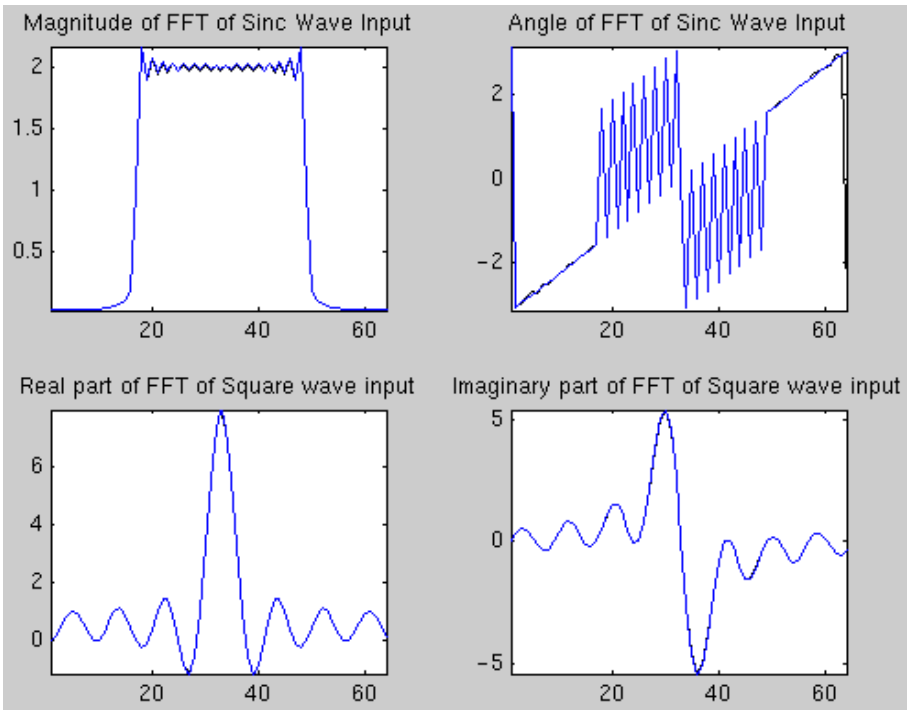


(i) N=8

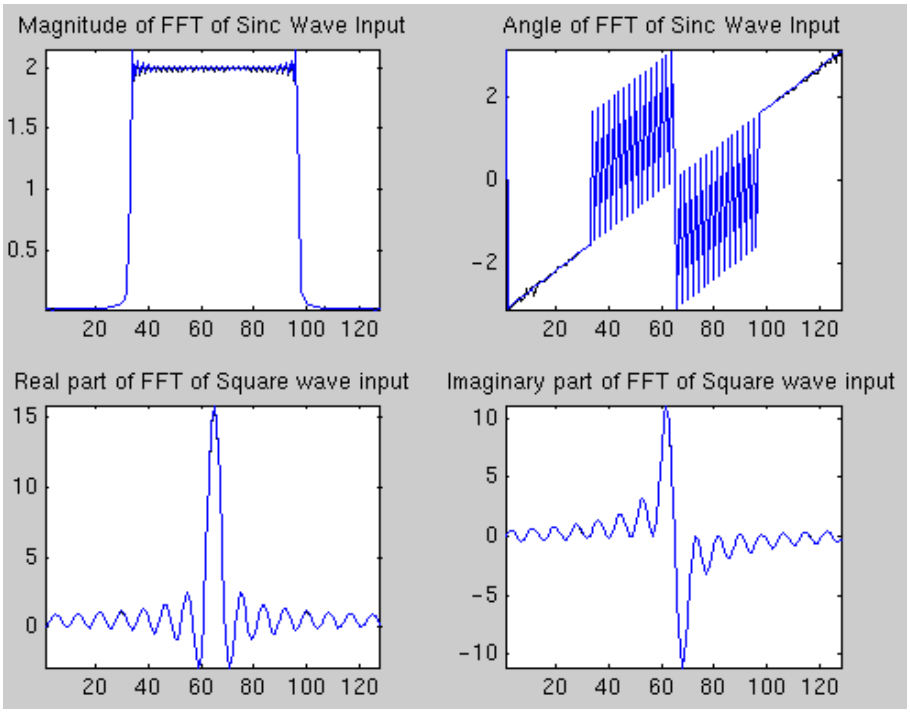


(ii) N=32

Figure 4-8: FFT Simulations for Different N: 8, 32

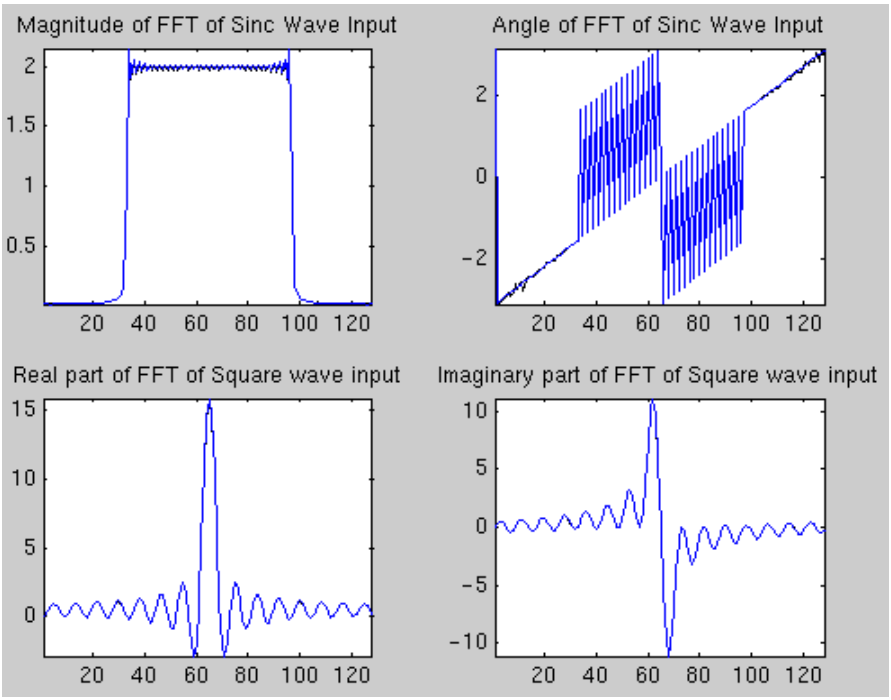


(i) N=64

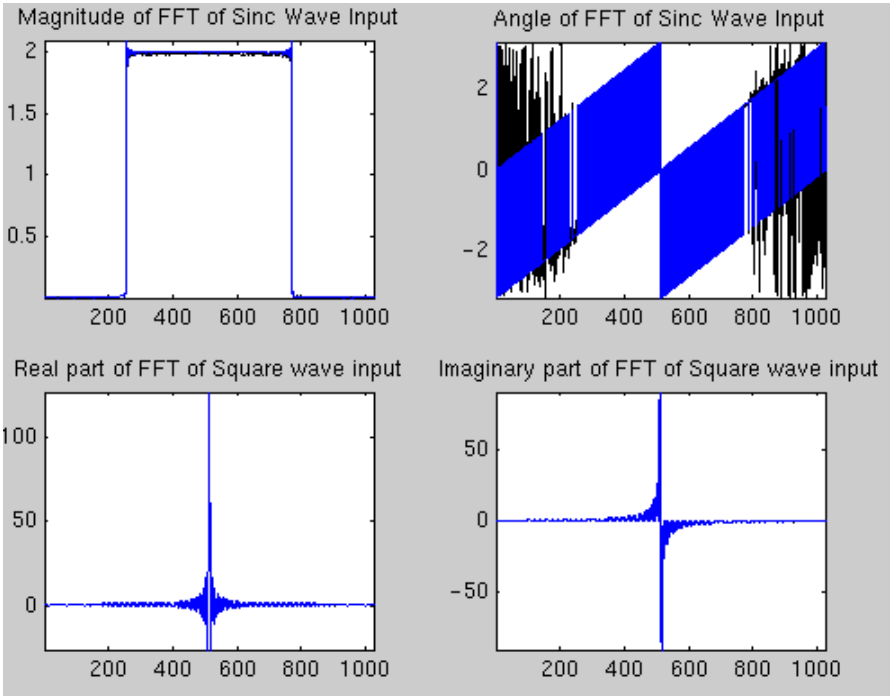


(ii) N=128

Figure 4-9: FFT Simulations for Different N: 64, 128



(i) N=256



(ii) N=1024

Figure 4-10: FFT Simulations for Different N: 256, 1024

Chapter 5: Conclusions and Future Work

The FFT algorithm was successfully created and tested through post-layout simulations. The macro has been created and tested successfully in hardware. A method for creating test vectors was created for different parameters, and the FFT was synthesized and tested for different numbers of points. The FFT was synthesized to different processes. The Rounder macros were each created and successfully simulated. The Fixed Rounder macro was successfully instantiated with different parameters.

By successfully designing and verifying the FFT, it was shown that designing a flexible macro of moderate complexity in VHDL was possible. Future studies involving this FFT might include comparing its capabilities and flexibility to other FFT algorithms. Implementing and testing the FFT in hardware for several different configurations could also be done. Additional work can be done to create more DSP modules and create a library of parameterized VHDL macros. Future research could also explore how much time is saved using this FFT algorithm compared to developing an FFT algorithm from scratch, and to modifying a non-parameterized version of the FFT to make it work with a different configuration.

There are several modifications that could be performed on the FFT code. Better default values could be created, or an exploration can be performed of how the bit growth for the multipliers and addition can be changed and what the optimum parameter values are for a given input width and desired output width. If a twiddle factor generator were created to replace the ROM based twiddle factor look-up, this algorithm could be expanded to larger sizes. User-controlled pipelining could be implemented also, with options for different pipelined configurations.

Overall this research was a success. By creating the FFT module, an example of a parameterized macro is available. By creating the Rounder macros, some simple examples of useful parameterized macros are available. By verifying the FFT, it can be shown that it was successfully implemented, and the verification process can be used as an example. The impact that these would have on design time would be significant if an engineer needs an FFT with specifications that are in the range of parameters this FFT has. By expanding the range of parameters, it becomes more likely this FFT will be reused and save design time.

List of References

[1] *Wikipedia: The Free Encyclopedia*. 15 Jan. 2001. 20 July 2003. <http://www.wikipedia.org/wiki/Main_Page>

[2] Ashenden, Peter J. *The Designer's Guide to VHDL, 2nd Edition*. Morgan Kaufmann, San Fransico, CA, 1996.

[3] Altium Limited. "The VHDL Language Guide" 2000. July 2002. < http://www.acc-eda.com/vhdlref/refguide/toclist.htm>

[4] Smith, Stephen W. *The Scientist and Engineer's Guide to Digital Signal Processing*. California Technical Publishing, 1997. August 2002. <http://www.dspguide.com/pdfbook.htm>

[5] Bores Signal Processing. "On-line DSP Class" *DSP Training* 2002. September 2002. < http://www.bores.com/index_online.htm>

[6] Taft, Jeffery D. "The Inverse FFT Page", *DSP Design Performance*. 2001. Oct. 2002. <http://www.nauticom.net/www/jdtaft/inverseFFT.htm>

[7] Chan, Patrick, Jason Mah, Andrew Sung, and Raymond Sung. "Linear Feedback Shift Register", *E552 Application Notes* December 1999. September 2002. <http://www.ee.ualberta.ca/~elliott/ee552/studentAppNotes/1999f/Drivers_Ed/lfsr.html>

[8] He, Shousheng and Mats Torkelson. "A New Approach to a Pipeline FFT Processor" 1996. August 2002. <http://ipdps.eece.unm.edu/1996/PAPERS/S19/HE/HE.PDF>

[9] Xilinx, Inc. "Xilinx – Reference Designs" September 2002. < http://www.xilinx.com/ipcenter/reference_designs/>

Appendices

Appendix A: Rounder Code

output_gain_stage.vhd

```
--The output gain stage gain adjust and rounds the result of a DSP
block
--so that it optimally loads the next block.

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE ieee.std_logic_signed.all;
USE ieee.std_logic_unsigned.all;

entity output_gain_stage is
      generic(input_width : integer :=25;
                  output_width : integer :=12;
                  rndr_sel_width : integer :=3;
                  gain_width : integer :=7;
                  cr_output_width : integer :=16;
                  cr_rad_pos : integer :=5;
                  fr_rad_pos : integer :=10;
                  ri : integer :=1;
                  ro : integer :=1;
                  rc : integer :=1;
                  pr : integer :=1;
                  rt : integer :=1);
      port (      d_in : in signed(input_width-1 downto 0);
                  q_out : out signed(output_width-1 downto 0);
                  cr_rndr_sel : in std_logic_vector(rndr_sel_width-1
downto 0);
                  gain : in unsigned(gain_width-1 downto 0);
                  clock : in std_logic;
                  reset_n : in std_logic;
                  out_load_enable : in std_logic);
end output_gain_stage;

architecture behavior of output_gain_stage is
constant fri : integer :=cr_output_width+gain_width;
signal gain_sgnd : signed(gain_width downto 0);
signal gain_reg : signed(gain_width downto 0);
signal cr_out : signed(cr_output_width-1 downto 0);
signal cr_out_test : signed(cr_output_width-1 downto 0);
signal mult_out : signed(fri downto 0);
signal fr_input : signed(fri-1 downto 0);

component config_rndr
      generic(
      rndr_sel_width : integer;       -- bit width of config rounder
select input
      input_width    : integer;       -- bit width of data input
      output_width   : integer;       -- bit width of data output
```

```
      cr_rad_pos     : integer;       -- radix position for config
rounder
        ri : integer;
        ro : integer;
        rc : integer;
        rt : integer);
      port(
      clock           : in std_logic;                                 --
clock
      reset_n       : in std_logic;                                 --
active low reset
      out_load_enable : in std_logic;
      d_in          : in signed(input_width-1 downto 0);        --
input data
      cr_rndr_sel   : in std_logic_vector(rndr_sel_width-1 downto 0);
-- mux select for configurable rounder
      q_out         : out signed(output_width-1 downto 0));     --
output data
end component;

component fixed_rndr
      generic( input_width : integer;
                  output_width : integer;
                  fr_rad_pos : integer;
                  ri: integer;
                  ro: integer;
                  rt: integer);
      port (      d_in : in signed(input_width-1 downto 0);
                  clock : in std_logic;
                  reset_n : in std_logic;
                  out_load_enable : in std_logic;
                  q_out : out signed(output_width-1 downto 0));
end component;

component plr
      generic(width : integer;
                  rt : integer);
      port (      clock : in std_logic;
                  reset_n : in std_logic;
                  d : in signed(width-1 downto 0);
                  q : out signed(width-1 downto 0));
end component;

begin
      gain_sgnd<=signed('0' & gain);
gain_reg_on : if rc=1 generate
      in_reg_gain : plr
            generic map (width=>gain_width+1, rt=>rt)
            port map  ( clock=>clock, reset_n=>reset_n,
                              d=>gain_sgnd, q=>gain_reg);
end generate gain_reg_on;

gain_reg_off : if rc/=1 generate
      gain_reg<=gain_sgnd;
```

```
end generate gain_reg_off;

cfg_rndr : config_rndr
      generic map (input_width=>input_width,
                        output_width=>cr_output_width,
                        rndr_sel_width=>rndr_sel_width,
                        cr_rad_pos=>cr_rad_pos,
                        ri=>ri, ro=>0, rc=>rc, rt=>rt)
      port map (  d_in=>d_in, q_out=>cr_out,
                        cr_rndr_sel=>cr_rndr_sel,
                        clock=>clock, reset_n=>reset_n,
                        out_load_enable=>out_load_enable);
pipe1_on : if pr=1 generate
      cr_piped: plr
            generic map(width=>cr_output_width, rt=>rt)
            port map (  clock=>clock, reset_n=>reset_n,
                              d=>cr_out, q=>cr_out_test);
end generate pipe1_on;

pipe1_off : if pr=0 generate
      cr_out_test<=cr_out;
end generate pipe1_off;

process(gain_reg, cr_out_test)
begin
      mult_out<=cr_out_test*gain_reg;
end process;
fr_input<=mult_out(fri-1 downto 0);

fix_rndr : fixed_rndr
      generic map (input_width=>fri,
                        output_width=>output_width,
                        fr_rad_pos=>fr_rad_pos,
                        ri=>0, ro=>ro, rt=>rt)
      port map (  d_in=>fr_input,
                        clock=>clock,
                        reset_n=>reset_n,
                        out_load_enable=>out_load_enable,
                        q_out=>q_out);
end;
```

config_rndr.vhd

```
--Configurable rounder
--I/O
--d_in : data input
--q_out : data output
--cr_rndr_sel : Padding (sign ext/lsb padding) select.
--
--Generics:
--input_width     :     bit width of the input
--output_width    :     bit width of the output
--rndr_sel_width: Padding amount select width (sign ext/lsb padding)
```

```
--cr_rad_pos      :     configurable rounder radix position.
--                            bit position of the input that you
--                            are rounding on.
--ri              :     Input register type - 0/1 none/plr
--ro              :     Output register type - 0/1/2 none/plr/z
--rc              :     configuration register type - 0/1 none/plr
--rt              :     Reset select for registers - 0/1/2
none/synch/asynch
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE ieee.std_logic_signed.all;
USE ieee.std_logic_unsigned.all;

entity config_rndr is
      generic(input_width : integer :=23;
                  output_width : integer :=12;
                  rndr_sel_width : integer :=2;
                  cr_rad_pos : integer :=11;
                  ri : integer :=0;
                  ro : integer :=1;
                  rc : integer :=0;
                  rt : integer :=1);
      port (      d_in : in signed(input_width-1 downto 0);
                  q_out : out signed(output_width-1 downto 0);
                  cr_rndr_sel : in std_logic_vector(rndr_sel_width-1
downto 0);
                  clock : in std_logic;
                  reset_n : in std_logic;
                  out_load_enable : in std_logic);
end config_rndr;

architecture behavior of config_rndr is
constant ws : integer:=rndr_sel_width;
constant wi : integer:=input_width;
signal d_in_reg : signed(wi-1 downto 0);
signal cr_temp : signed(ws-1 downto 0);
signal cr_rndr_sel_reg : signed(ws-1 downto 0);
signal cr_rndr_sel_stdl : std_logic_vector(ws-1 downto 0);
type extension_array is array (0 to 2**ws-1) of signed(wi+2**ws-2
downto 0);
signal ext_d_in : extension_array;
signal fixed_rndr_input : signed(wi+2**ws-2 downto 0);


component plr
      generic(width : integer;
                  rt : integer);
      port (      clock : in std_logic;
                  reset_n : in std_logic;
                  d : in signed(width-1 downto 0);
                  q : out signed(width-1 downto 0));
end component;
```

```
component zreg
      generic(width : integer;
                  rt : integer);
      port (      clock : in std_logic;
                  reset_n : in std_logic;
                  out_load_enable : in std_logic;
                  d : in signed(width-1 downto 0);
                  q : out signed(width-1 downto 0));
end component;

component fixed_rndr
      generic( input_width : integer;
                  output_width : integer;
                  fr_rad_pos : integer;
                  ri: integer;
                  ro: integer;
                  rt: integer);
      port (      d_in : in signed(input_width-1 downto 0);
                  clock : in std_logic;
                  reset_n : in std_logic;
                  out_load_enable : in std_logic;
                  q_out : out signed(output_width-1 downto 0));
end component;

begin
d_in_reg_on : if ri=1 generate
      in_reg_d : plr
            generic map (width=>input_width, rt=>rt)
            port map (  clock=>clock, reset_n=>reset_n,
                              d=>d_in, q=>d_in_reg);
end generate d_in_reg_on;

d_in_reg_off : if ri/=1 generate
      d_in_reg<=d_in;
end generate d_in_reg_off;

cr_reg_on : if rc=1 generate
      cr_temp<=signed(cr_rndr_sel);
      in_reg_cr : plr
            generic map (width=>rndr_sel_width, rt=>rt)
            port map (  clock=>clock, reset_n=>reset_n,
                              d=>cr_temp, q=>cr_rndr_sel_reg);
      cr_rndr_sel_stdl<=std_logic_vector(cr_rndr_sel_reg);
end generate cr_reg_on;

cr_reg_off : if rc/=1 generate
      cr_rndr_sel_stdl<=cr_rndr_sel;
end generate;

process(d_in_reg)
begin
      for i in 0 to 2**ws-1 loop
            for j in (wi+2**ws-2) downto 0 loop
                  if j>(wi+i-1) then
```

```
                        ext_d_in(i)(j)<=d_in_reg(wi-1);
                  else
                        if j<i then
                              ext_d_in(i)(j)<='0';
                        else
                              ext_d_in(i)(j)<=d_in_reg(j-i);
                        end if;
                  end if;
            end loop;
      end loop;
end process;

process(cr_rndr_sel_stdl,ext_d_in)
begin
      fixed_rndr_input<=ext_d_in(conv_integer(unsigned(cr_rndr_sel_stdl
)));
end process;

fixed_rounder: fixed_rndr
      generic map (input_width=>wi+2**ws-1,
                        output_width=>output_width,
                        fr_rad_pos=>cr_rad_pos,
                        ri=>0, ro=>ro, rt=>rt)
      port map (  d_in=>fixed_rndr_input,
                        clock=>clock,
                        reset_n=>reset_n,
                        out_load_enable=>out_load_enable,
                        q_out=>q_out);


end;
```

fixed_rndr.vhd

```
--Fixed rounder
--Takes an input of input_width bits wide, rounds it
--  at fr_rad_pos radix position (0 is no rounding),
--  then converts it to output_width bits wide by
--  padding or clipping.
--
--input_width:  bit width of input, can be any number greater than 1.
--output_width:  bit width of output, must be greater than 1.
--fr_rad_pos:  integer value, radix position of input, position that is
--             being rounded.
--
--I/O
--d_in:  data input
--q_out:  data output
--
--Generics:
--input_width     :     bit width of the input
--output_width    :     bit width of the output
--fr_rad_pos      :     fixed rounder radix position.
```

```
--                              bit position of the input that you
--                              are rounding on.
--ri             :     Input register type - 0/1 none/plr
--ro             :     Output register type - 0/1/2 none/plr/z
--rt             :     Reset select for registers - 0/1/2
none/synch/asynch

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE ieee.std_logic_signed.all;

entity fixed_rndr is
      generic( input_width : integer :=23;
                  output_width : integer :=12;
                  fr_rad_pos : integer :=10;
                  ri: integer :=0;
                  ro: integer :=0;
                  rt: integer :=1);
      port (      d_in : in signed(input_width-1 downto 0);
                  clock : in std_logic;
                  reset_n : in std_logic;
                  out_load_enable : in std_logic;
                  q_out : out signed(output_width-1 downto 0));
end fixed_rndr;

architecture behavior of fixed_rndr is
signal d_in_reg: signed(input_width-1 downto 0);
signal q_out_reg: signed(output_width-1 downto 0);
signal rounded: signed(input_width-fr_rad_pos-1 downto 0);
--constant overflowcheck : integer :=2**(input_width-abs(fr_rad_pos)-
1)-1;
--constant negcheck : integer :=2**(abs(input_width-fr_rad_pos-
output_width)+1)-1;
function ones(n: integer) return unsigned is
      variable slv: unsigned(n-1 downto 0);
begin
      slv:=(others=>'1');
      return slv;
end ones;

component plr
      generic(width : integer;
                  rt : integer);
      port (      clock : in std_logic;
                  reset_n : in std_logic;
                  d : in signed(width-1 downto 0);
                  q : out signed(width-1 downto 0));
end component;

component zreg
      generic(width : integer;
                  rt : integer);
      port (      clock : in std_logic;
```

```
                  reset_n : in std_logic;
                  out_load_enable : in std_logic;
                  d : in signed(width-1 downto 0);
                  q : out signed(width-1 downto 0));
end component;

begin
in_reg_on: if ri=1 generate
      in_reg: plr
            generic map (width=>input_width, rt=>rt)
            port map (  clock=>clock, reset_n=>reset_n,
                              d=>d_in, q=>d_in_reg);
end generate in_reg_on;

in_reg_off : if ri/=1 generate
      d_in_reg<=d_in;
end generate in_reg_off;

      process(d_in_reg)
      begin
            if (fr_rad_pos<0) then
                  rounded(input_width-fr_rad_pos-1 downto
abs(fr_rad_pos))<=d_in_reg;
                  rounded(0-fr_rad_pos-1 downto 0)<=(others=>'0');
            else
                  if (fr_rad_pos=0) then
                        rounded<=d_in_reg;
                  else
                        if unsigned(d_in_reg(input_width-1 downto
fr_rad_pos))=ones(input_width-fr_rad_pos-1) then
                              rounded<=d_in_reg(input_width-1 downto
fr_rad_pos);
                        else
                              if (fr_rad_pos=1) then

      rounded<=signed(d_in_reg(input_width-1 downto
fr_rad_pos))+d_in_reg(fr_rad_pos-1);
                              else
                                    if (unsigned(d_in_reg(fr_rad_pos-2
downto 0))=conv_unsigned(0,fr_rad_pos-1)) and (d_in_reg(input_width-
1)='1') then

      rounded<=d_in_reg(input_width-1 downto fr_rad_pos);
                                    else

      rounded<=signed(d_in_reg(input_width-1 downto
fr_rad_pos))+d_in_reg(fr_rad_pos-1);
                                    end if;
                              end if;
                        end if;
                  end if;
            end if;
      end process;
```

```
extending: if (input_width-fr_rad_pos)<=output_width generate
      q_out_reg(input_width-fr_rad_pos-1 downto 0)<=rounded;
      q_out_reg(output_width-1 downto (input_width-
fr_rad_pos))<=(others=>rounded(input_width-fr_rad_pos-1));
end generate extending;
clipping: if (input_width-fr_rad_pos)>output_width generate
      process(rounded)
      begin
            if rounded(input_width-fr_rad_pos-1)='0' then
                  if unsigned(rounded(input_width-fr_rad_pos-1 downto
output_width-1))/=conv_unsigned(0,input_width-fr_rad_pos-
output_width+1) then
                        q_out_reg(output_width-1)<='0';
                        for i in output_width-2 downto 0 loop
                              q_out_reg(i)<='1';
                        end loop;
                  else
                        q_out_reg<=rounded(output_width-1 downto 0);
                  end if;
            else
                  if unsigned(rounded(input_width-fr_rad_pos-1 downto
output_width-1))/=ones(input_width-fr_rad_pos-output_width+1) then
                        q_out_reg(output_width-1)<='1';
                        q_out_reg(0)<='1';
                        if output_width>2 then
                              for i in output_width-2 downto 1 loop
                                    q_out_reg(i)<='0';
                              end loop;
                        end if;
                  else
                        if unsigned(rounded(output_width-2 downto
0))=conv_unsigned(0,output_width-1) then
                              q_out_reg(output_width-1)<='1';
                              q_out_reg(0)<='1';
                              if output_width>2 then
                                    for i in output_width-2 downto 1
loop
                                          q_out_reg(i)<='0';
                                    end loop;
                              end if;
                        else
                              q_out_reg<=rounded(output_width-1 downto
0);
                        end if;
                  end if;
            end if;
      end process;
end generate clipping;

out_reg_plr: if ro=1 generate
      out_reg: plr
            generic map (width=>output_width, rt=>rt)
            port map (  clock=>clock, reset_n=>reset_n,
                        d=>q_out_reg, q=>q_out);
```

```
end generate out_reg_plr;

out_reg_z: if ro=2 generate
      out_reg: zreg
            generic map (width=>output_width, rt=>rt)
            port map (  clock=>clock, reset_n=>reset_n,
                              out_load_enable=>out_load_enable,
                              d=>q_out_reg, q=>q_out);
end generate out_reg_z;

out_reg_off : if ro/=1 and ro/=2 generate
      q_out<=q_out_reg;
end generate out_reg_off;
end;
```

plr.vhd

```
--Pipline Register of n bits.
--only 1 stage deep
--signed type data
--
--Reset is set by:
--  0 - no reset
--  1 - synchronous reset
--  2 - asynchronous reset
--Reset is active low

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE ieee.std_logic_signed.all;

entity plr is
      generic(width : integer:=10;
                  rt : integer:=1);
      port (      clock : in std_logic;
                  reset_n : in std_logic;
                  d : in signed(width-1 downto 0);
                  q : out signed(width-1 downto 0));
end plr;

architecture behavior of plr is

begin
      none: if rt=0 generate
      process(clock)
            begin
                  if clock'event and clock='1' then
                        q<=d;
                  end if;
            end process;
      end generate none;

      asynch: if rt=2 generate
```

```
            process(clock,reset_n)
            begin
                  if reset_n='0' then
                        q<=(others=>'0');
                  else
                        if clock'event and clock='1' then
                              q<=d;
                        end if;
                  end if;
            end process;
      end generate asynch;

      synch: if rt/=0 and rt/=2 generate
            process(clock)
            begin
                  if clock'event and clock='1' then
                        if reset_n='0' then
                              q<=(others=>'0');
                        else
                              q<=d;
                        end if;
                  end if;
            end process;
      end generate synch;
end;
```

zreg.vhd

```
--Pipline Register of n bits.
--only 1 stage deep
--signed type data
--
--Reset is set by:
--  0  - no reset
--  1  - synchronous reset
--  2  - asynchronous reset
--Reset is active low, defaults to synch

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE ieee.std_logic_signed.all;

entity zreg is
      generic(width : integer:=10;
                  rt : integer:=1);
      port (      clock : in std_logic;
                  reset_n : in std_logic;
                  out_load_enable : in std_logic;
                  d : in signed(width-1 downto 0);
                  q : out signed(width-1 downto 0));
end zreg;

architecture behavior of zreg is
```

```
signal old_q : signed(width-1 downto 0);
begin
      none: if rt=0 generate
            process(clock)
            begin
                  if clock'event and clock='1' then
                        if out_load_enable='1' then
                              q<=d;
                              old_q<=d;
                        else
                              q<=old_q;
                        end if;
                  end if;
            end process;
      end generate none;

      asynch: if rt=2 generate
            process(clock,reset_n)
            begin
                  if reset_n='0' then
                        q<=(others=>'0');
                        old_q<=(others=>'0');
                  else
                        if clock'event and clock='1' then
                              if out_load_enable='1' then
                                    q<=d;
                                    old_q<=d;
                              else
                                    q<=old_q;
                              end if;
                        end if;
                  end if;
            end process;
      end generate asynch;

      synch: if rt/=0 and rt/=2 generate
            process(clock)
            begin
                  if clock'event and clock='1' then
                        if reset_n='0' then
                              q<=(others=>'0');
                              old_q<=(others=>'0');
                        else
                              if out_load_enable='1' then
                                    q<=d;
                                    old_q<=d;
                              else
                                    q<=old_q;
                              end if;
                        end if;
                  end if;
            end process;
      end generate synch;
end;
```

Appendix B: FFT Code

fft_filed_tb.vhd

```
--File I/O version of the test bench
--reads in a file called testvec, outputs to a file called data.out.
--Only handles serial input/output.
--Only needs a clock input, outputs the fft output in bit reversed
order.

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE std.textio.ALL;
USE work.fft_pkg.all;


entity fft_filed_tb is
      generic (   N : integer :=64; --number of points
                        m : integer :=12; --input width
                        add_g : integer :=1; --adder growth
                        mult_g : integer :=9; --Growth of the
multipliers;
                        twiddle_width : integer :=10 --width of twiddle
factor roms
                  );
      port (      clock : in std_logic;
                  Xout_r : out std_logic_vector (m+((log2(N)-
1)/2)*mult_g+log2(N)*add_g-1 downto 0);
                  Xout_i : out std_logic_vector (m+((log2(N)-
1)/2)*mult_g+log2(N)*add_g-1 downto 0)
                  );
end fft_filed_tb;

architecture stateflow of fft_filed_tb is

-- converts a character into a std_logic

function char_to_stdl(c: character) return std_logic is
    variable sl: std_logic;
    begin
      case c is
        when 'U' =>
           sl := 'U';
        when 'X' =>
           sl := 'X';
        when '0' =>
           sl := '0';
        when '1' =>
           sl := '1';
        when 'Z' =>
           sl := 'Z';
```

```
        when 'W' =>
           sl := 'W';
        when 'L' =>
           sl := 'L';
        when 'H' =>
           sl := 'H';
        when '-' =>
           sl := '-';
        when others =>
           sl := 'X';
    end case;
   return sl;
  end char_to_stdl;


-- converts a string into std_logic_vector
function str_to_stdvec(s: string) return std_logic_vector is
  variable slv: std_logic_vector(s'high-s'low downto 0);
  variable k: integer;
begin
   k := s'high-s'low;
  for i in s'range loop
     slv(k) := char_to_stdl(s(i));
     k      := k - 1;
  end loop;
  return slv;
end str_to_stdvec;

--converts a std_logic_vector to a string
function stdvec_to_str(inp: std_logic_vector) return string is
variable temp: string(inp'left+1 downto 1) := (others => 'X');
begin
    for i in inp'reverse_range loop
        if (inp(i) = '1') then
            temp(i+1) := '1';
        elsif (inp(i) = '0') then
            temp(i+1) := '0';
        end if;
    end loop;
    return temp;
end;-- function stdvec_to_str;
signal resetn : std_logic;
signal load_enable : std_logic;
signal xrsi : std_logic_vector (m-1 downto 0);
signal xisi : std_logic_vector (m-1 downto 0);
signal outdata_r : std_logic_vector (m+((log2(N)-
1)/2)*mult_g+log2(N)*add_g-1 downto 0);
signal outdata_i : std_logic_vector (m+((log2(N)-
1)/2)*mult_g+log2(N)*add_g-1 downto 0);

component fft
      generic (   N : integer;
                        m : integer;
                        add_g : integer;
```

```
                        mult_g : integer;
                        twiddle_width : integer);
      port (      clk : in std_logic;
                  resetn : in std_logic;
                  load_enable : in std_logic;
                  xrsi : in std_logic_vector(m-1 downto 0);
                  xisi : in std_logic_vector(m-1 downto 0);
                  Xrso : out std_logic_vector (m+((log2(N)-
1)/2)*mult_g+log2(N)*add_g-1 downto 0);
                  Xiso : out std_logic_vector (m+((log2(N)-
1)/2)*mult_g+log2(N)*add_g-1 downto 0)
                  );
end component;

begin
fft_core : fft
      generic map (N=>N, m=>m, add_g=>add_g, mult_g=>mult_g,
twiddle_width=>twiddle_width)
      port map (  clk=>clock,resetn=>resetn,load_enable=>load_enable,
                        xrsi=>xrsi, xisi=>xisi,
                        Xrso=>outdata_r, Xiso=>outdata_i);
process(outdata_r,outdata_i)
begin
      Xout_r<=outdata_r;
      Xout_i<=outdata_i;
end process;

process(clock)
      file my_output : TEXT open WRITE_MODE is "data.out";
      file my_input : TEXT open READ_MODE is "testvec";
      variable my_data, file_line :LINE;
      variable stimulus_in : STRING (m*2+2 downto 1);
      variable data_out : string ((m+((log2(N)-
1)/2)*mult_g+log2(N)*add_g)*2 downto 1);
      variable stim_stdl : std_logic_vector(m*2+1 downto 0);
begin
      if ((clock'event and clock='0') and not endfile(my_input)) then
            data_out:=stdvec_to_str(outdata_r) &
stdvec_to_str(outdata_i);
            write(my_data,data_out);
            writeline(my_output,my_data);
            readline(my_input,file_line);
            read(file_line,stimulus_in);
            stim_stdl:=str_to_stdvec(stimulus_in);
      elsif ((clock'event and clock='1') and not endfile(my_input))
then
            resetn<=stim_stdl(m*2+1);
            load_enable<=stim_stdl(m*2);
            xrsi<=stim_stdl(m*2-1 downto m);
            xisi<=stim_stdl(m-1 downto 0);
      end if;
end process;
```
end stateflow;

fft.vhd

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
use work.fft_pkg.all;

entity fft is
      generic (   N : integer :=1024;--number of samples
                        Serial_in : integer :=1; --1/0 serial/parallel
in
                        Serial_out: integer :=1; --1/0 serial/parallel
out
                        Output_order_in_natural: integer :=0; --1/0
output in natural/bit rev. order
                        m : integer :=12; --bit width of input
                        add_g : integer :=1; --Growth during adders, 1
or 0
                        mult_g : integer :=4; --growth during
multipliers, up to twiddle_width+1
                        twiddle_width : integer :=10
                        );
      port (      --parameter-dependent ports
                  xrsi : in std_logic_vector(m-1 downto 0) :=(others
=>'0');
                  xisi : in std_logic_vector(m-1 downto 0) :=(others
=>'0');
--                xrpi : in ioarray(0 to N-1) :=(others =>(m-1 downto 0
=>'0'));
--                xipi : in ioarray(0 to N-1) :=(others =>(m-1 downto 0
=>'0'));
                  Xrso : out std_logic_vector(m+((log2(N)-
1)/2)*mult_g+log2(N)*add_g-1 downto 0);
                  Xiso : out std_logic_vector(m+((log2(N)-
1)/2)*mult_g+log2(N)*add_g-1 downto 0);
--                Xrpo : out ioarray(0 to N-1) :=(others
=>(m+((log2(N)-1)/2)*mult_g+log2(N)*add_g-1 downto 0 =>'0'));
--                Xipo : out ioarray(0 to N-1) :=(others
=>(m+((log2(N)-1)/2)*mult_g+log2(N)*add_g-1 downto 0 =>'0'));
                  --required ports
                  clk : in std_logic;
                  load_enable : in std_logic;
                  resetn : in std_logic
                  );
end fft;

architecture structure of fft is
constant num_stages : integer :=log2(N);
constant w : integer := m+((num_stages-1)/2)*mult_g+num_stages*add_g;
signal incore_r : std_logic_vector(m-1 downto 0);
signal incore_i : std_logic_vector(m-1 downto 0);
signal outcore_r : std_logic_vector(w-1 downto 0);
signal outcore_i : std_logic_vector(w-1 downto 0);
```

```
component fft_core
      generic (input_width: integer;
                  twiddle_width: integer;
                  N : integer;
                  add_g : integer;
                  mult_g : integer);
      port (      clock : in std_logic;
                  resetn : in std_logic;
                  load_enable : in std_logic;
                  xin_r : in std_logic_vector(input_width-1 downto 0);
                  xin_i : in std_logic_vector(input_width-1 downto 0);
                  Xout_r : out std_logic_vector (input_width+((log2(N)-
1)/2)*mult_g+log2(N)*add_g-1 downto 0);
                  Xout_i : out std_logic_vector (input_width+((log2(N)-
1)/2)*mult_g+log2(N)*add_g-1 downto 0)
                  );
end component;

component s2pconv
      generic (width: integer;
                  N: integer;
                  Output_order_in_natural: integer
                  );
      port (      clock : in std_logic;
                  load_enable : in std_logic;
                  resetn : in std_logic;
                  pdata_r : out ioarray(0 to N-1);
                  pdata_i : out ioarray(0 to N-1);
                  Xout_r : in std_logic_vector (width-1 downto 0);
                  Xout_i : in std_logic_vector (width-1 downto 0)
                  );
end component;

component p2sconv
      generic (input_width : integer; N : integer);
      port (      clock : in std_logic;
                  load_enable : in std_logic;
                  resetn : in std_logic;
                  pdata_r : in ioarray(0 to N-1);
                  pdata_i : in ioarray(0 to N-1);
                  xin_r : out std_logic_vector (input_width-1 downto
0);
                  xin_i : out std_logic_vector (input_width-1 downto 0)
                  );
end component;

begin

fft_main: fft_core
      generic map (input_width=>m, twiddle_width=>twiddle_width, N=>N,
                        add_g=>add_g,mult_g=>mult_g)
      port map (  clock=>clk, resetn=>resetn, load_enable=>load_enable,
                        xin_r=>incore_r, xin_i=>incore_i,
                        Xout_r=>outcore_r, Xout_i=>outcore_i
```

```
                        );

s_in : if Serial_in=1 generate
      incore_r<=xrsi;
      incore_i<=xisi;
end generate;

--p_in : if Serial_in=0 generate
--    input_conv : p2sconv
--          generic map (input_width=>m, N=>N)
--          port map (  clock=>clk, load_enable=>load_enable,
resetn=>resetn,
--                              pdata_r=>xrpi,
--                              pdata_i=>xipi,
--                              xin_r=>incore_r, xin_i=>incore_i);
--end generate;

s_out : if Serial_out=1 generate
      Xrso<=outcore_r;
      Xiso<=outcore_i;
end generate;

--p_out : if Serial_out=0 generate
--    output_conv : s2pconv
--          generic map (width=>w, N=>N,
Output_order_in_natural=>Output_order_in_natural)
--          port map (  clock=>clk, load_enable=>load_enable,
resetn=>resetn,
--                              pdata_r=>xrpo,
--                              pdata_i=>xipo,
--                              Xout_r=>outcore_r, Xout_i=>outcore_i);
--end generate;

end;
```

fft_core.vhd

```
-- N point FFT
-- Uses R2^2SDF algorithm
--
-- Generics used:
-- N - number of points taken - powers of 2 only, ranging from 8 to
1024 points.
-- input_width - bit width of the input vector
-- twiddle width - width of the twiddle factors stored in the ROM
-- add_g - Adder growth - Adders grow 0 or 1 bits each time they are
used
--         Exculdes adders in the complex multiplier (that is handled
by mult_g)
-- mult_g - multiplier growth - 0 to twiddle_width+1 - Growth during
the complex
--                multiplier stages
--
```

```
-- Width of output vector is as follows (num_stages=log2(N):
--    N                 width
--   8,16         input_width + (num_stages * add_g) + mult_g
--    32,64       input_width + (num_stages * add_g) + 2*mult_g
-- 128,256        input_width + (num_stages * add_g) + 3*mult_g
-- 512,1024       input_width + (num_stages * add_g) + 4*mult_g
--
-- Due to the way this system was made parameterizable, there are many
signals
-- that will remain unconnected.  This is normal.
--
-- Default generics are for a simple 64 point FFT

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
use work.fft_pkg.all;

entity fft_core is
      generic (   input_width : integer :=12;
                        twiddle_width : integer :=10;
                        N : integer :=64;
                        add_g : integer :=1;  --Either 0 or 1 only.
                        mult_g : integer :=9  --Can be any number from
0 to twiddle_width+1
                  );
      port (      clock : in std_logic;
                  resetn : in std_logic;
                  load_enable : in std_logic;
                  xin_r : in std_logic_vector(input_width-1 downto 0);
                  xin_i : in std_logic_vector(input_width-1 downto 0);
                  Xout_r : out std_logic_vector (input_width+((log2(N)-
1)/2)*mult_g+log2(N)*add_g-1 downto 0);
                  Xout_i : out std_logic_vector (input_width+((log2(N)-
1)/2)*mult_g+log2(N)*add_g-1 downto 0)
                  );
end fft_core;

architecture structure of fft_core is
--Signal declarations
constant num_stages: integer :=log2(N);
signal control: std_logic_vector(num_stages-1 downto 0);
type stage_array is array (1 to num_stages-1) of
      std_logic_vector(input_width+(num_stages*add_g)+(((num_stages-
1)/2)*mult_g)-1 downto 0);
signal stoscon_r: stage_array;
signal stoscon_i: stage_array;
type rom_array is array (1 to (num_stages-1)/2) of
std_logic_vector(twiddle_width-1 downto 0);
signal rtoscon_r: rom_array;
signal rtoscon_i: rom_array;
--component declarations
component counterhle
      generic (width : integer);
```

```
      port (      clock : in std_logic;
                  resetn : in std_logic;
                  load_enable : in std_logic;
                  countout : out std_logic_vector(width-1 downto 0)
                  );
end component;

component rom1
      generic (data_width : integer; address_width : integer);
      port  (address    : IN     std_logic_vector (address_width-1
DOWNTO 0);
                  datar        : OUT    std_logic_vector (data_width-1
DOWNTO 0);
                  datai        : OUT    std_logic_vector (data_width-1
DOWNTO 0)
                  );
end component;

component rom2
      generic (data_width : integer; address_width : integer);
      port  (address    : IN     std_logic_vector (address_width-1
DOWNTO 0);
                  datar        : OUT    std_logic_vector (data_width-1
DOWNTO 0);
                  datai        : OUT    std_logic_vector (data_width-1
DOWNTO 0)
                  );
end component;

component rom3
      generic (data_width : integer; address_width : integer);
      port  (address    : IN     std_logic_vector (address_width-1
DOWNTO 0);
                  datar        : OUT    std_logic_vector (data_width-1
DOWNTO 0);
                  datai        : OUT    std_logic_vector (data_width-1
DOWNTO 0)
                  );
end component;

component rom4
      generic (data_width : integer; address_width : integer);
      port  (address    : IN     std_logic_vector (address_width-1
DOWNTO 0);
                  datar        : OUT    std_logic_vector (data_width-1
DOWNTO 0);
                  datai        : OUT    std_logic_vector (data_width-1
DOWNTO 0)
                  );
end component;

component stage_I
      generic     (data_width : INTEGER; add_g : INTEGER; shift_stages
: INTEGER);
```

```
      port  (prvs_r :in std_logic_vector(data_width-1-add_g downto 0);
                  prvs_i :in std_logic_vector(data_width-1-add_g downto
0);
                  s :in std_logic; clock : in std_logic; resetn : in
std_logic;
                  tonext_r :out std_logic_vector(data_width-1 downto
0);
                  tonext_i :out std_logic_vector(data_width-1 downto
0));
end component;

component stage_II
      generic     (data_width : INTEGER; add_g : INTEGER; mult_g :
INTEGER;
                  twiddle_width : INTEGER; shift_stages : INTEGER);
      port  (prvs_r :in std_logic_vector(data_width-1-add_g downto 0);
                  prvs_i :in std_logic_vector(data_width-1-add_g downto
0);
                  t :in std_logic; s :in std_logic; clock : in
std_logic;
                  resetn : in std_logic;
                  fromrom_r :in std_logic_vector(twiddle_width-1 downto
0);
                  fromrom_i :in std_logic_vector(twiddle_width-1 downto
0);
                  tonext_r :out std_logic_vector(data_width+mult_g-1
downto 0);
                  tonext_i :out std_logic_vector(data_width+mult_g-1
downto 0));
end component;

component stage_I_last
      generic     (data_width : INTEGER; add_g : INTEGER);
      port  (prvs_r :in std_logic_vector(data_width-1-add_g downto 0);
                  prvs_i :in std_logic_vector(data_width-1-add_g downto
0);
                  s :in std_logic; clock : in std_logic; resetn : in
std_logic;
                  tonext_r :out std_logic_vector(data_width-1 downto
0);
                  tonext_i :out std_logic_vector(data_width-1 downto
0));
end component;

component stage_II_last
      generic     (data_width : INTEGER; add_g : INTEGER);
      port  (prvs_r :in std_logic_vector(data_width-1-add_g downto 0);
                  prvs_i :in std_logic_vector(data_width-1-add_g downto
0);
                  t :in std_logic; s :in std_logic; clock : in
std_logic;
                  resetn : in std_logic;
                  tonext_r :out std_logic_vector(data_width-1 downto
0);
```

```
                  tonext_i :out std_logic_vector(data_width-1 downto
0));
end component;

begin
controller : component counterhle
      generic map (width=>num_stages)
      port map (  clock=>clock,resetn=>resetn,load_enable=>load_enable,
                        countout=>control);
stages : for i in 1 to num_stages generate
--    constant parity : integer :=i rem 2;
--    constant shift_stages : integer := 2**(num_stages - i);
--    constant rom_loc : integer :=i/2;
--    constant data_width : integer :=input_width + (i*add_g) + (((i-
1)/2)*mult_g);
--    constant s: integer :=(num_stages-i);
--    constant t: integer :=(num_stages-i+1);
      initial_stage: if i=1 generate
            initial_stage_I :  component stage_I
                  generic map (data_width=>input_width + (i*add_g) +
(((i-1)/2)*mult_g),
                                    add_g=>add_g,
shift_stages=>2**(num_stages - i))
                  port map (
      prvs_r=>xin_r,prvs_i=>xin_i,s=>control((num_stages-
i)),clock=>clock,resetn=>resetn,
                                    tonext_r=>stoscon_r(i)(input_width
+ (i*add_g) + (((i-1)/2)*mult_g)-1 downto 0),
                                    tonext_i=>stoscon_i(i)(input_width
+ (i*add_g) + (((i-1)/2)*mult_g)-1 downto 0));
      end generate initial_stage;

      even_stages: if ((i rem 2)=0) and (i/=num_stages) generate
            inner_stage_II : component stage_II
                  generic map (data_width=>input_width + (i*add_g) +
(((i-1)/2)*mult_g),
                                    add_g=>add_g,mult_g=>mult_g,
                                    twiddle_width=>twiddle_width,
shift_stages=>2**(num_stages - i))
                  port map (  prvs_r=>stoscon_r(i-1)(input_width +
(i*add_g) + (((i-1)/2)*mult_g)-1-add_g downto 0),
                                    prvs_i=>stoscon_i(i-1)(input_width
+ (i*add_g) + (((i-1)/2)*mult_g)-1-add_g downto 0),
                                    t=>control((num_stages-
i+1)),s=>control((num_stages-i)),clock=>clock,resetn=>resetn,

      fromrom_r=>rtoscon_r(i/2),fromrom_i=>rtoscon_i(i/2),
                                    tonext_r=>stoscon_r(i)(input_width
+ (i*add_g) + (((i-1)/2)*mult_g)+mult_g-1 downto 0),
                                    tonext_i=>stoscon_i(i)(input_width
+ (i*add_g) + (((i-1)/2)*mult_g)+mult_g-1 downto 0));

            first_rom: if (i/2)=1 generate
                  rom_1 : component rom1
```

```
                        generic map (data_width=>twiddle_width,
address_width=>(num_stages-i+1)+1)
                        port map (  address=>control((num_stages-i+1)
downto 0),

      datar=>rtoscon_r(i/2),datai=>rtoscon_i(i/2));
            end generate first_rom;

            second_rom: if (i/2)=2 generate
                  rom_2 : component rom2
                        generic map (data_width=>twiddle_width,
address_width=>(num_stages-i+1)+1)
                        port map (  address=>control((num_stages-i+1)
downto 0),

      datar=>rtoscon_r(i/2),datai=>rtoscon_i(i/2));
            end generate second_rom;

            third_rom: if (i/2)=3 generate
                  rom_3 : component rom3
                        generic map (data_width=>twiddle_width,
address_width=>(num_stages-i+1)+1)
                        port map (  address=>control((num_stages-i+1)
downto 0),

      datar=>rtoscon_r(i/2),datai=>rtoscon_i(i/2));
            end generate third_rom;

            fourth_rom: if (i/2)=4 generate
                  rom_4 : component rom4
                        generic map (data_width=>twiddle_width,
address_width=>(num_stages-i+1)+1)
                        port map (  address=>control((num_stages-i+1)
downto 0),

      datar=>rtoscon_r(i/2),datai=>rtoscon_i(i/2));
            end generate fourth_rom;

      end generate even_stages;

      odd_stages: if (((i rem 2)=1) and (i/=num_stages)) and (i/=1)
      generate
            inner_stage_I : component stage_I
                  generic map (data_width=>input_width + (i*add_g) +
(((i-1)/2)*mult_g),
                                    add_g=>add_g,
shift_stages=>2**(num_stages - i))
                  port map (  prvs_r=>stoscon_r(i-1)(input_width +
(i*add_g) + (((i-1)/2)*mult_g)-1-add_g downto 0),
                                    prvs_i=>stoscon_i(i-1)(input_width
+ (i*add_g) + (((i-1)/2)*mult_g)-1-add_g downto 0),
                                    s=>control((num_stages-
i)),clock=>clock,resetn=>resetn,
```

```
                                    tonext_r=>stoscon_r(i)(input_width
+ (i*add_g) + (((i-1)/2)*mult_g)-1 downto 0),
                                    tonext_i=>stoscon_i(i)(input_width
+ (i*add_g) + (((i-1)/2)*mult_g)-1 downto 0));
      end generate odd_stages;

      end_on_even: if (i=num_stages) and ((i rem 2)=0) generate
            last_stage_II : component stage_II_last
                  generic map (data_width=>input_width + (i*add_g) +
(((i-1)/2)*mult_g), add_g=>add_g)
                  port map (  prvs_r=>stoscon_r(i-1)(input_width +
(i*add_g) + (((i-1)/2)*mult_g)-1-add_g downto 0),
                                    prvs_i=>stoscon_i(i-1)(input_width
+ (i*add_g) + (((i-1)/2)*mult_g)-1-add_g downto 0),
                                    t=>control((num_stages-
i+1)),s=>control((num_stages-i)),clock=>clock,resetn=>resetn,
                                    tonext_r=>Xout_r,tonext_i=>Xout_i);
      end generate end_on_even;

      end_on_odd: if (i=num_stages) and ((i rem 2)=1) generate
            last_stage_I : component stage_I_last
                  generic map (data_width=>input_width + (i*add_g) +
(((i-1)/2)*mult_g), add_g=>add_g)
                  port map (  prvs_r=>stoscon_r(i-1)(input_width +
(i*add_g) + (((i-1)/2)*mult_g)-1-add_g downto 0),
                                    prvs_i=>stoscon_i(i-1)(input_width
+ (i*add_g) + (((i-1)/2)*mult_g)-1-add_g downto 0),
                                    s=>control((num_stages-
i)),clock=>clock,resetn=>resetn,
                                    tonext_r=>Xout_r,tonext_i=>Xout_i);


      end generate end_on_odd;

end generate stages;
end;
```

counterhle.vhd

```
--Counter with resetn and load enable
--When load enable is high, it counts.
--When load enable is low, it stops counting.
--When a reset is triggered, it resets to zero.

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;

entity counterhle is
      generic (   width: integer :=3);
      port (      clock : in std_logic;
                  resetn : in std_logic;
                  load_enable : in std_logic;
```

```
                countout : out std_logic_vector(width-1 downto 0);
                hle : out std_logic
                );
end counterhle;

architecture behavior of counterhle is
signal count : std_logic_vector(width-1 downto 0);
signal hold_load_enable : std_logic;
begin
process(clock)
begin
      if (resetn='0')then
         count <= (others => '0');
         hold_load_enable <='0';
      elsif (clock'event and clock='1') then
                if (load_enable = '1' or hold_load_enable='1') then
                      count <= unsigned(count) + '1';
                else
                      count <= (others =>'0');
                end if;
                if (unsigned(count)+'1')=0 then
                      hold_load_enable <= load_enable;
                end if;
      end if;
end process;
countout <= count;
hle <=hold_load_enable;
end;
```

stage_I.vhd

```
--Component for Stages using BF2I (first and every odd stage)
--Doesn't handle last stage
--Input is a standard logic vector of data_width-add_g
--data_width - width of the internal busses
--add_g - Add growth variable - if 1, data_width grows by 1, if 0 then
0
--shift_stages - number of shift register stages

library IEEE;
use IEEE.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity stage_I is
generic     (     data_width : INTEGER :=13;
                  add_g : INTEGER := 1;
                  shift_stages : INTEGER := 32
            );
port  (     prvs_r :in std_logic_vector(data_width-1-add_g downto 0);
                  prvs_i :in std_logic_vector(data_width-1-add_g downto
0);
                  s :in std_logic;
                  clock : in std_logic;
```

```
                resetn : in std_logic;
                tonext_r :out std_logic_vector(data_width-1 downto
0);
                tonext_i :out std_logic_vector(data_width-1 downto 0)
            );
end stage_I;

architecture structure of stage_I is
signal toreg_r : std_logic_vector(data_width-1 downto 0);
signal toreg_i : std_logic_vector(data_width-1 downto 0);
signal fromreg_r : std_logic_vector(data_width-1 downto 0);
signal fromreg_i : std_logic_vector(data_width-1 downto 0);

component shiftregN
      generic (data_width : integer;
             n : integer);
      port  (clock : IN std_logic;
                read_data  : OUT    std_logic_vector (data_width-1
DOWNTO 0);
                write_data : IN     std_logic_vector (data_width-1
DOWNTO 0);
                resetn     : IN     std_logic
                );
end component;

component BF2I
      generic (data_width : INTEGER;
                add_g: INTEGER);
      port        (fromreg_r :in std_logic_vector(data_width-1 downto
0);
                fromreg_i :in std_logic_vector(data_width-1 downto
0);
                prvs_r :in std_logic_vector(data_width-add_g-1 downto
0);
                prvs_i :in std_logic_vector(data_width-add_g-1 downto
0);
                s : in std_logic;
                toreg_r :out std_logic_vector(data_width-1 downto 0);

                toreg_i :out std_logic_vector(data_width-1 downto 0);
                tonext_r :out std_logic_vector(data_width-1 downto
0);
                tonext_i :out std_logic_vector(data_width-1 downto 0)
                );
end component;

begin
regr : shiftregN
      generic map (data_width=>data_width, n=>shift_stages)
      port map (clock=>clock, read_data=>fromreg_r,
write_data=>toreg_r, resetn=>resetn);
regi : shiftregN
      generic map (data_width=>data_width, n=>shift_stages)
```

```
      port map (clock=>clock, read_data=>fromreg_i,
write_data=>toreg_i, resetn=>resetn);
btrfly : BF2I
      generic map (data_width=>data_width, add_g=>add_g)
      port map (  fromreg_r=>fromreg_r, fromreg_i=>fromreg_i,
                        prvs_r=>prvs_r, prvs_i=>prvs_i,
                        s=>s,
                        toreg_r=>toreg_r, toreg_i=>toreg_i,
                        tonext_r=>tonext_r, tonext_i=>tonext_i);
end;
```

stage_I_last.vhd

```
--Component for Stages using BF2I (if BF2I is the last stage)
--Input is a standard logic vector of data_width-add_g
--data_width - width of the internal busses
--add_g - Add growth variable - if 1, data_width grows by 1, if 0 then
0
--Only 1 shift stage

library IEEE;
use IEEE.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity stage_I_last is
generic     (     data_width : INTEGER :=13;
                  add_g : INTEGER := 1
            );
port  (     prvs_r :in std_logic_vector(data_width-1-add_g downto 0);
                  prvs_i :in std_logic_vector(data_width-1-add_g downto
0);
                  s :in std_logic;
                  clock : in std_logic;
                  resetn : in std_logic;
                  tonext_r :out std_logic_vector(data_width-1 downto
0);
                  tonext_i :out std_logic_vector(data_width-1 downto 0)
            );
end stage_I_last;

architecture structure of stage_I_last is
signal toreg_r : std_logic_vector(data_width-1 downto 0);
signal toreg_i : std_logic_vector(data_width-1 downto 0);
signal fromreg_r : std_logic_vector(data_width-1 downto 0);
signal fromreg_i : std_logic_vector(data_width-1 downto 0);

component shiftreg1
      generic (data_width : integer);
      port  (clock : IN std_logic;
                  read_data  : OUT    std_logic_vector (data_width-1
DOWNTO 0);
                  write_data : IN     std_logic_vector (data_width-1
DOWNTO 0);
```

```
                  resetn     : IN     std_logic
                  );
end component;

component BF2I
      generic (data_width : INTEGER;
                  add_g: INTEGER);
      port        (fromreg_r :in std_logic_vector(data_width-1 downto
0);
                  fromreg_i :in std_logic_vector(data_width-1 downto
0);
                  prvs_r :in std_logic_vector(data_width-add_g-1 downto
0);
                  prvs_i :in std_logic_vector(data_width-add_g-1 downto
0);
                  s : in std_logic;
                  toreg_r :out std_logic_vector(data_width-1 downto 0);

                  toreg_i :out std_logic_vector(data_width-1 downto 0);
                  tonext_r :out std_logic_vector(data_width-1 downto
0);
                  tonext_i :out std_logic_vector(data_width-1 downto 0)
                  );
end component;

begin
regr : shiftreg1
      generic map (data_width=>data_width)
      port map (clock=>clock, read_data=>fromreg_r,
write_data=>toreg_r, resetn=>resetn);
regi : shiftreg1
      generic map (data_width=>data_width)
      port map (clock=>clock, read_data=>fromreg_i,
write_data=>toreg_i, resetn=>resetn);
btrfly : BF2I
      generic map (data_width=>data_width, add_g=>add_g)
      port map (  fromreg_r=>fromreg_r, fromreg_i=>fromreg_i,
                        prvs_r=>prvs_r, prvs_i=>prvs_i,
                        s=>s,
                        toreg_r=>toreg_r, toreg_i=>toreg_i,
                        tonext_r=>tonext_r, tonext_i=>tonext_i);
end;
```

stage_II.vhd

```
--Component for Stages using BF2II (second and every even stage)
--Doesn't handle last stage
--Input is a standard logic vector of data_width-add_g
--data_width - width of the internal busses
--add_g - Add growth variable - if 1, data_width grows by 1, if 0 then
0
--mult_g - mult growth variable - can range from 0 to twiddle_width+1
--twiddle_width - width of the twiddle factor input
--shift_stages - number of shift register stages
```

```
library IEEE;
use IEEE.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity stage_II is
generic     (     data_width : INTEGER :=14;
                  add_g : INTEGER := 1;
                  mult_g : INTEGER :=9;
                  twiddle_width : INTEGER :=10;
                  shift_stages : INTEGER := 16
            );

port  (     prvs_r :in std_logic_vector(data_width-1-add_g downto 0);
                  prvs_i :in std_logic_vector(data_width-1-add_g downto
0);
                  t :in std_logic;
                  s :in std_logic;
                  clock : in std_logic;
                  resetn : in std_logic;
                  fromrom_r :in std_logic_vector(twiddle_width-1 downto
0);
                  fromrom_i :in std_logic_vector(twiddle_width-1 downto
0);
                  tonext_r :out std_logic_vector(data_width+mult_g-1
downto 0);
                  tonext_i :out std_logic_vector(data_width+mult_g-1
downto 0)
            );
end stage_II;

architecture structure of stage_II is
signal toreg_r : std_logic_vector(data_width-1 downto 0);
signal toreg_i : std_logic_vector(data_width-1 downto 0);
signal fromreg_r : std_logic_vector(data_width-1 downto 0);
signal fromreg_i : std_logic_vector(data_width-1 downto 0);
signal tomult_r : std_logic_vector(data_width-1 downto 0);
signal tomult_i : std_logic_vector(data_width-1 downto 0);

component shiftregN
      generic (data_width : integer;
             n : integer);
      port  (clock : IN std_logic;
                  read_data  : OUT    std_logic_vector (data_width-1
DOWNTO 0);
                  write_data : IN     std_logic_vector (data_width-1
DOWNTO 0);
                  resetn     : IN     std_logic
                  );
end component;

component BF2II
      generic (data_width : INTEGER;
```

```
                  add_g: INTEGER);
      port        (fromreg_r :in std_logic_vector(data_width-1 downto
0);
                  fromreg_i :in std_logic_vector(data_width-1 downto
0);
                  prvs_r :in std_logic_vector(data_width-add_g-1 downto
0);
                  prvs_i :in std_logic_vector(data_width-add_g-1 downto
0);
                  t : in std_logic;
                  s : in std_logic;
                  toreg_r :out std_logic_vector(data_width-1 downto 0);

                  toreg_i :out std_logic_vector(data_width-1 downto 0);
                  tonext_r :out std_logic_vector(data_width-1 downto
0);
                  tonext_i :out std_logic_vector(data_width-1 downto 0)
                  );
end component;

component twiddle_mult
      generic (mult_width : INTEGER;
             twiddle_width : INTEGER;
            output_width : INTEGER);
      port  (data_r :in std_logic_vector(mult_width-1 downto 0);
                  data_i :in std_logic_vector(mult_width-1 downto 0);
                  twdl_r :in std_logic_vector(twiddle_width-1 downto
0);
                  twdl_i :in std_logic_vector(twiddle_width-1 downto
0);
                  out_r :out std_logic_vector(output_width-1 downto 0);

                  out_i :out std_logic_vector(output_width-1 downto 0)
                  );
end component;

begin
regr : shiftregN
      generic map (data_width=>data_width, n=>shift_stages)
      port map (clock=>clock, read_data=>fromreg_r,
write_data=>toreg_r, resetn=>resetn);
regi : shiftregN
      generic map (data_width=>data_width, n=>shift_stages)
      port map (clock=>clock, read_data=>fromreg_i,
write_data=>toreg_i, resetn=>resetn);
btrfly : BF2II
      generic map (data_width=>data_width, add_g=>add_g)
      port map (  fromreg_r=>fromreg_r, fromreg_i=>fromreg_i,
                        prvs_r=>prvs_r, prvs_i=>prvs_i,
                        t=>t, s=>s,
                        toreg_r=>toreg_r, toreg_i=>toreg_i,
                        tonext_r=>tomult_r, tonext_i=>tomult_i);
twiddle : twiddle_mult
```

```
      generic map (mult_width=>data_width,
twiddle_width=>twiddle_width,
                        output_width=> data_width+mult_g)
      port map (  data_r=>tomult_r, data_i=>tomult_i,
                        twdl_r=>fromrom_r, twdl_i=>fromrom_i,
                        out_r=>tonext_r, out_i=>tonext_i);
end;
```

stage_II_last.vhd

```
--Component for Stages using BF2II (last stage only)
--When BF2II is the last butterfly, there is no multiplier after it
--Input is a standard logic vector of data_width-add_g
--data_width - width of the internal busses
--add_g - Add growth variable - if 1, data_width grows by 1, if 0 then
0

library IEEE;
use IEEE.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity stage_II_last is
generic     (     data_width : INTEGER :=14;
                  add_g : INTEGER := 1
            );

port  (     prvs_r :in std_logic_vector(data_width-1-add_g downto 0);
                  prvs_i :in std_logic_vector(data_width-1-add_g downto
0);
                  t :in std_logic;
                  s :in std_logic;
                  clock : in std_logic;
                  resetn : in std_logic;
                  tonext_r :out std_logic_vector(data_width-1 downto
0);
                  tonext_i :out std_logic_vector(data_width-1 downto 0)
            );
end stage_II_last;

architecture structure of stage_II_last is
signal toreg_r : std_logic_vector(data_width-1 downto 0);
signal toreg_i : std_logic_vector(data_width-1 downto 0);
signal fromreg_r : std_logic_vector(data_width-1 downto 0);
signal fromreg_i : std_logic_vector(data_width-1 downto 0);

component shiftreg1
      generic (data_width : integer);
      port  (clock : IN std_logic;
                  read_data  : OUT    std_logic_vector (data_width-1
DOWNTO 0);
                  write_data : IN     std_logic_vector (data_width-1
DOWNTO 0);
                  resetn     : IN     std_logic
                  );
```

```
end component;

component BF2II
      generic (data_width : INTEGER;
                  add_g: INTEGER);
      port        (fromreg_r :in std_logic_vector(data_width-1 downto
0);
                  fromreg_i :in std_logic_vector(data_width-1 downto
0);
                  prvs_r :in std_logic_vector(data_width-add_g-1 downto
0);
                  prvs_i :in std_logic_vector(data_width-add_g-1 downto
0);
                  t : in std_logic;
                  s : in std_logic;
                  toreg_r :out std_logic_vector(data_width-1 downto 0);

                  toreg_i :out std_logic_vector(data_width-1 downto 0);
                  tonext_r :out std_logic_vector(data_width-1 downto
0);
                  tonext_i :out std_logic_vector(data_width-1 downto 0)
                  );
end component;

begin
regr : shiftreg1
      generic map (data_width=>data_width)
      port map (clock=>clock, read_data=>fromreg_r,
write_data=>toreg_r, resetn=>resetn);
regi : shiftreg1
      generic map (data_width=>data_width)
      port map (clock=>clock, read_data=>fromreg_i,
write_data=>toreg_i, resetn=>resetn);
btrfly : BF2II
      generic map (data_width=>data_width, add_g=>add_g)
      port map (  fromreg_r=>fromreg_r, fromreg_i=>fromreg_i,
                        prvs_r=>prvs_r, prvs_i=>prvs_i,
                        t=>t, s=>s,
                        toreg_r=>toreg_r, toreg_i=>toreg_i,
                        tonext_r=>tonext_r, tonext_i=>tonext_i);
end;
```

BF2I.vhd

```
--Butterfly stage type 1
--7/17/02

library IEEE;
use IEEE.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity BF2I is

generic (   data_width : INTEGER :=13;
```

```
                add_g : INTEGER :=1
        );

port        (     fromreg_r :in std_logic_vector(data_width-1 downto
0);
                  fromreg_i :in std_logic_vector(data_width-1 downto
0);
                  prvs_r :in std_logic_vector(data_width-1-add_g downto
0);
                  prvs_i :in std_logic_vector(data_width-1-add_g downto
0);
                  s : in std_logic;
                  toreg_r :out std_logic_vector(data_width-1 downto 0);

                  toreg_i :out std_logic_vector(data_width-1 downto 0);
                  tonext_r :out std_logic_vector(data_width-1 downto
0);
                  tonext_i :out std_logic_vector(data_width-1 downto 0)
            );

end BF2I;

architecture behavior of  BF2I is
signal prvs_ext_r : std_logic_vector(data_width-1 downto 0);
signal prvs_ext_i : std_logic_vector(data_width-1 downto 0);
signal add1_out : std_logic_vector(data_width downto 0);
signal add2_out : std_logic_vector(data_width downto 0);
signal sub1_out : std_logic_vector(data_width downto 0);
signal sub2_out : std_logic_vector(data_width downto 0);

component adder
      generic (inst_width:integer);
        port(
             inst_A : in std_logic_vector(data_width-1 downto 0);
             inst_B : in std_logic_vector(data_width-1 downto 0);
             SUM : out std_logic_vector(data_width downto 0)
             );
end component;

component subtract
      generic (inst_width:integer);
        port(
            inst_A : in std_logic_vector(data_width-1 downto 0);
            inst_B : in std_logic_vector(data_width-1 downto 0);
            DIFF : out std_logic_vector(data_width downto 0)
            );
end component;

component mux2_mmw
      generic (data_width:integer);
            port(
                  s : in std_logic;
                  in0: in std_logic_vector(data_width-1 downto 0);
                  in1: in std_logic_vector(data_width downto 0);
```

```
                  data: out std_logic_vector(data_width-1 downto 0)
                  );
end component;

begin

      add1 : adder
      generic map (inst_width=>data_width)
      port map (inst_A=>prvs_ext_r, inst_B=>fromreg_r, SUM=>add1_out);

      add2 : adder
      generic map (inst_width=>data_width)
      port map (inst_A=>prvs_ext_i, inst_B=>fromreg_i, SUM=>add2_out);

      sub1 : subtract
      generic map (inst_width=>data_width)
      port map (inst_A=>fromreg_r, inst_B=>prvs_ext_r, DIFF=>sub1_out);

      sub2 : subtract
      generic map (inst_width=>data_width)
      port map (inst_A=>fromreg_i, inst_B=>prvs_ext_i, DIFF=>sub2_out);

      mux_1 : mux2_mmw
      generic map (data_width=>data_width)
      port map (s=>s, in0=>fromreg_r, in1=>add1_out, data=>tonext_r);

      mux_2 : mux2_mmw
      generic map (data_width=>data_width)
      port map (s=>s, in0=>fromreg_i, in1=>add2_out, data=>tonext_i);

      mux_3 : mux2_mmw
      generic map (data_width=>data_width)
      port map (s=>s, in0=>prvs_ext_r, in1=>sub1_out, data=>toreg_r);

      mux_4 : mux2_mmw
      generic map (data_width=>data_width)
      port map (s=>s, in0=>prvs_ext_i, in1=>sub2_out, data=>toreg_i);

process(prvs_r, prvs_i)
begin
      if add_g=1 then
            prvs_ext_r <= prvs_r(data_width-2) &  prvs_r;
            prvs_ext_i <= prvs_i(data_width-2) &  prvs_i;
      else
            prvs_ext_r <= prvs_r;
            prvs_ext_i <= prvs_i;
      end if;
end process;

end;
```

BF2II.vhd

```
--Butterfly stage type 2
--7/17/02

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity BF2II is

generic (   data_width : INTEGER :=13;
                  add_g: INTEGER :=1
        );

port        (     fromreg_r :in std_logic_vector(data_width-1 downto
0);
                  fromreg_i :in std_logic_vector(data_width-1 downto
0);
                  prvs_r :in std_logic_vector(data_width-1-add_g downto
0);
                  prvs_i :in std_logic_vector(data_width-1-add_g downto
0);
                  t : in std_logic;
                  s : in std_logic;
                  toreg_r :out std_logic_vector(data_width-1 downto 0);

                  toreg_i :out std_logic_vector(data_width-1 downto 0);
                  tonext_r :out std_logic_vector(data_width-1 downto
0);
                  tonext_i :out std_logic_vector(data_width-1 downto 0)
            );

end BF2II;

architecture behavior of  BF2II is
signal prvs_ext_r : std_logic_vector(data_width-1 downto 0);
signal prvs_ext_i : std_logic_vector(data_width-1 downto 0);
signal add1_out : std_logic_vector(data_width downto 0);
signal add2_out : std_logic_vector(data_width downto 0);
signal sub1_out : std_logic_vector(data_width downto 0);
signal sub2_out : std_logic_vector(data_width downto 0);
signal swapadd    : std_logic_vector(data_width downto 0);
signal swapsub    : std_logic_vector(data_width downto 0);

component  adder
      generic (inst_width:integer);
        port(
             inst_A : in std_logic_vector(data_width-1 downto 0);
             inst_B : in std_logic_vector(data_width-1 downto 0);
             SUM : out std_logic_vector(data_width downto 0)
             );
end component;
```

```
component subtract
      generic (inst_width:integer);
        port(
            inst_A : in std_logic_vector(data_width-1 downto 0);
            inst_B : in std_logic_vector(data_width-1 downto 0);
            DIFF : out std_logic_vector(data_width downto 0)
            );
end component;

component mux2_mmw
      generic (data_width:integer);
            port(
                  s : in std_logic;
                  in0: in std_logic_vector(data_width-1 downto 0);
                  in1: in std_logic_vector(data_width downto 0);
                  data: out std_logic_vector(data_width-1 downto 0)
                  );
end component;

begin

      add1 : adder
      generic map (inst_width=>data_width)
      port map (inst_A=>prvs_ext_r, inst_B=>fromreg_r, SUM=>add1_out);

      add2 : adder
      generic map (inst_width=>data_width)
      port map (inst_A=>prvs_ext_i, inst_B=>fromreg_i, SUM=>add2_out);

      sub1 : subtract
      generic map (inst_width=>data_width)
      port map (inst_A=>fromreg_r, inst_B=>prvs_ext_r, DIFF=>sub1_out);

      sub2 : subtract
      generic map (inst_width=>data_width)
      port map (inst_A=>fromreg_i, inst_B=>prvs_ext_i, DIFF=>sub2_out);

      mux_1 : mux2_mmw
      generic map (data_width=>data_width)
      port map (s=>s, in0=>fromreg_r, in1=>add1_out, data=>tonext_r);

      mux_2 : mux2_mmw
      generic map (data_width=>data_width)
      port map (s=>s, in0=>fromreg_i, in1=>swapadd, data=>tonext_i);

      mux_3 : mux2_mmw
      generic map (data_width=>data_width)
      port map (s=>s, in0=>prvs_ext_r, in1=>sub1_out, data=>toreg_r);

      mux_4 : mux2_mmw
      generic map (data_width=>data_width)
      port map (s=>s, in0=>prvs_ext_i, in1=>swapsub, data=>toreg_i);
```

```
process(prvs_r,prvs_i,s,t)
begin
      if add_g=1 then
            if (t='0' and s='1') then
                  prvs_ext_r <= prvs_i(data_width-2) & prvs_i;
                  prvs_ext_i <= prvs_r(data_width-2) & prvs_r;
            else
                  prvs_ext_r <= prvs_r(data_width-2) & prvs_r;
                  prvs_ext_i <= prvs_i(data_width-2) & prvs_i;
            end if;
      else
            if (t='0' and s='1') then
                  prvs_ext_r <= prvs_i;
                  prvs_ext_i <= prvs_r;
            else
                  prvs_ext_r <= prvs_r;
                  prvs_ext_i <= prvs_i;
            end if;
      end if;
end process;

process(add2_out, sub2_out, s, t)
begin
      if (t='0' and s='1') then
            swapadd<=sub2_out;
            swapsub<=add2_out;
      else
            swapadd<=add2_out;
            swapsub<=sub2_out;
      end if;
end process;
end;
```

twiddle_mult.vhd

```
--Twiddle multiplier
--7/17/02
--Uses a both inputs same width complex multiplier
--Won't sign extend output, but will truncate it down
--(mult_width + twiddle_width >= output_width)
--
--Twiddle factors are limited to -1 < twdl < 1.
--(twiddle factor can't be 0b1000000000)

library IEEE;
use IEEE.std_logic_1164.all;

entity twiddle_mult is

generic (
            mult_width : INTEGER := 7;
            twiddle_width : INTEGER :=3;
            output_width : INTEGER :=9
```

```
        );

port        (     data_r :in std_logic_vector(mult_width-1 downto 0);
                  data_i :in std_logic_vector(mult_width-1 downto 0);
                  twdl_r :in std_logic_vector(twiddle_width-1 downto
0);
                  twdl_i :in std_logic_vector(twiddle_width-1 downto
0);
                  out_r :out std_logic_vector(output_width-1 downto 0);

                  out_i :out std_logic_vector(output_width-1 downto 0)
            );

end twiddle_mult;

architecture behavior of  twiddle_mult is
signal mult_out_r : std_logic_vector(twiddle_width + mult_width downto
0);
signal mult_out_i : std_logic_vector(twiddle_width + mult_width downto
0);

component comp_mult
      generic (   inst_width1:integer;
                  inst_width2:integer     );
      port  ( Re1  : in std_logic_vector(inst_width1-1 downto 0);
              Im1  : in std_logic_vector(inst_width1-1 downto 0);
              Re2  : in std_logic_vector(inst_width2-1 downto 0);
              Im2  : in std_logic_vector(inst_width2-1 downto 0);
              Re   : out std_logic_vector(inst_width1 + inst_width2
downto 0);
              Im   : out std_logic_vector(inst_width1 + inst_width2
downto 0)

            );

end component;

begin

      U1 : comp_mult
      generic map(
            inst_width1 => mult_width, inst_width2 => twiddle_width)
      port map (Re1=>data_r, Im1=>data_i, Re2=>twdl_r, Im2=>twdl_i,
Re=>mult_out_r, Im=>mult_out_i);

      process(mult_out_r,mult_out_i)
            begin
               out_r <= mult_out_r((twiddle_width+mult_width-1) downto
(twiddle_width+mult_width-output_width));
               out_i <= mult_out_i((twiddle_width+mult_width-1) downto
(twiddle_width+mult_width-output_width));

            end process;
end;
```

shiftregN.vhd

```
-- hds header_start
--n stage shift register, data_width bits wide.

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;


ENTITY shiftregN IS
   GENERIC(
      data_width : integer := 25;
      n      : integer := 4
   );
   PORT(
      clock      : IN     std_logic;
      read_data  : OUT    std_logic_vector (data_width-1 DOWNTO 0);
      write_data : IN     std_logic_vector (data_width-1 DOWNTO 0);
      resetn     : IN     std_logic
   );

-- Declarations

END shiftregN ;

-- hds interface_end
ARCHITECTURE behavior OF shiftregN IS
subtype reg is std_logic_vector(data_width-1 downto 0);
type regArray is array (integer range <>) of reg;

signal registerFile : regArray(1 to n-1);
BEGIN
process(Clock, resetn)
variable i: integer;
begin
      if (resetn='0') then
            for i in 1 to  (n-1) loop
                  registerFile(i) <= (others => '0');
                  read_data <= (others => '0');

            end loop;
      elsif (Clock'event and Clock='1') then
                  registerFile(1) <= write_data;
                  for i in 2 to n-1 loop
                   registerFile(i) <= registerFile(i-1);
                  end loop;
                  read_data <= registerFile(n-1);

      end if;
end process;

END behavior;
```

shiftreg1.vhd

```
-- hds header_start
--1 stage shift register, data_width bits wide.

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;


ENTITY shiftreg1 IS
   GENERIC(
      data_width : integer := 25
   );
   PORT(
      clock      : IN     std_logic;
      read_data  : OUT    std_logic_vector (data_width-1 DOWNTO 0);
      write_data : IN     std_logic_vector (data_width-1 DOWNTO 0);
      resetn     : IN     std_logic
   );

-- Declarations

END shiftreg1 ;

-- hds interface_end
ARCHITECTURE behavior OF shiftreg1 IS
--signal reg00 : std_logic_vector(data_width-1 downto 0);
BEGIN
process(Clock)
begin
      if (Clock'event and Clock='1') then
            if (resetn='0') then
--                for i in data_width-1 downto 0 loop
--                      reg00(i)<='0';
                      read_data <= (others => '0');
--                end loop;
            else
--                reg00<=write_data;
--                read_data<=reg00;
                  read_data  <=     write_data;

            end if;
      end if;
end process;
END behavior;
```

mux2_mmw.vhd

```
--Special 2 to 1 mux (mismatched width)
--7/17/02
--First input is data_width bits
--Second input is data_width+1 bits
--Ignores highest bit of second input
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;


ENTITY mux2_mmw IS
   GENERIC( data_width : integer := 35
                  );
   PORT     (     s : in std_logic;
                  in0: in std_logic_vector(data_width-1 downto 0);
                  in1: in std_logic_vector(data_width downto 0);
                  data: out std_logic_vector(data_width-1 downto 0)
                  );
END mux2_mmw ;

-- hds interface_end
ARCHITECTURE behavior OF mux2_mmw IS
BEGIN
process(in0,in1,s)
begin
      if s='0' then
            data<=in0;
      else
            data<=in1(data_width-1 downto 0);
      end if;
end process;
END behavior;
```

comp_mult.vhd

```
--Since both the real and imaginary values have the same number of
fractional bits,
--  there is no need to truncate.

library IEEE;
use IEEE.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity comp_mult is

generic (   inst_width1 : INTEGER := 14;
          inst_width2 : INTEGER := 14
        );

port  ( Re1  : in std_logic_vector(inst_width1-1 downto 0);
        Im1  : in std_logic_vector(inst_width1-1 downto 0);
        Re2  : in std_logic_vector(inst_width2-1 downto 0);
        Im2  : in std_logic_vector(inst_width2-1 downto 0);
        Re   : out std_logic_vector(inst_width1 + inst_width2 downto
0);
        Im   : out std_logic_vector(inst_width1 + inst_width2 downto 0)

);
```

```
end comp_mult;

architecture behavior of  comp_mult is
--multiplier outputs
signal product1 :std_logic_vector(inst_width1+inst_width2-1 downto 0);-
-re1*re2
signal product2 :std_logic_vector(inst_width1+inst_width2-1 downto 0);-
-im1*im2
signal product3 :std_logic_vector(inst_width1+inst_width2-1 downto 0);
signal product4 :std_logic_vector(inst_width1+inst_width2-1 downto 0);



component adder
      generic (inst_width:integer);
        port (
              inst_A : in std_logic_vector(inst_width-1 downto 0);
              inst_B : in std_logic_vector(inst_width-1 downto 0);
              SUM : out std_logic_vector(inst_width downto 0)
              );

end component;


component subtract
      generic (inst_width:integer);
        port (
             inst_A : in std_logic_vector(inst_width-1 downto 0);
             inst_B : in std_logic_vector(inst_width-1 downto 0);
             DIFF : out std_logic_vector(inst_width downto 0)
             );
end component;


component multiplier
      generic (inst_width1:integer;
            inst_width2:integer
            );
            port (
              inst_A : in std_logic_vector(inst_width1-1 downto 0);
              inst_B : in std_logic_vector(inst_width2-1 downto 0);
              PRODUCT_inst : out
std_logic_vector(inst_width1+inst_width2-1 downto 0)
             );
end component;


begin

        U1 : multiplier
           generic map( inst_width1=>inst_width1,
inst_width2=>inst_width2)
```

```
             port map ( inst_A => Re1, inst_B => Re2, PRODUCT_inst =>
product1 );

        U2 : multiplier
          generic map( inst_width1=>inst_width1,
inst_width2=>inst_width2)
            port map( inst_A => Im1, inst_B => Im2, PRODUCT_inst =>
product2 );

        U3 : multiplier
          generic map( inst_width1=>inst_width1,
inst_width2=>inst_width2)
             port map( inst_A => Re1, inst_B => Im2, PRODUCT_inst =>
product3);

        U4 : multiplier
          generic map( inst_width1=>inst_width2,
inst_width2=>inst_width1)
             port map ( inst_A => Re2, inst_B => Im1, PRODUCT_inst =>
product4);

        U5 : subtract
               generic map ( inst_width=>inst_width1+inst_width2)
             port map(inst_A => product1, inst_B => product2, DIFF =>
Re );

        U6 : adder
               generic map ( inst_width=>inst_width1+inst_width2)
             port map ( inst_A => product3, inst_B => product4, SUM =>
Im );

end;
```

adder.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity adder is
      generic (
            inst_width : INTEGER := 32
            );
      port (
            inst_A : in std_logic_vector(inst_width-1 downto 0);
            inst_B : in std_logic_vector(inst_width-1 downto 0);
            SUM : out std_logic_vector(inst_width downto 0)

            );
    end adder;

architecture oper of adder is
 signal a_signed, b_signed, sum_signed: SIGNED(inst_width downto 0);
```

```
begin
      a_signed <= SIGNED(inst_A(inst_width-1) & inst_A);
      b_signed <= SIGNED(inst_B(inst_width-1) & inst_B);
        sum_signed <= a_signed + b_signed;
      SUM <= std_logic_vector(sum_signed);
end oper;
```

subtract.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity subtract is
      generic (
            inst_width : INTEGER := 32
            );
      port (
            inst_A : in std_logic_vector(inst_width-1 downto 0);
            inst_B : in std_logic_vector(inst_width-1 downto 0);
            DIFF : out std_logic_vector(inst_width downto 0)

            );
    end subtract;

architecture oper of subtract is
 signal a_signed, b_signed, diff_signed: SIGNED(inst_width downto 0);

begin
      a_signed <= SIGNED(inst_A(inst_width-1) & inst_A);
      b_signed <= SIGNED(inst_B(inst_width-1) & inst_B);
        diff_signed <= a_signed - b_signed;
      DIFF <= std_logic_vector(diff_signed);
end oper;
```

multiplier.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity multiplier is
      generic (
            inst_width1 : INTEGER := 16;
            inst_width2 : INTEGER := 16
            );
      port (
            inst_A : in std_logic_vector(inst_width1-1 downto 0);
            inst_B : in std_logic_vector(inst_width2-1 downto 0);
            PRODUCT_inst : out std_logic_vector(inst_width1 +
inst_width2 - 1 downto 0)

            );
```

```
    end multiplier;

architecture oper of multiplier is
      signal mult_sig : SIGNED(inst_width1+inst_width2-1 downto 0) ;

begin
      mult_sig <= SIGNED(inst_A) * SIGNED(inst_B);
        PRODUCT_inst <= std_logic_vector(mult_sig);
end oper;
```

fft_pkg.vhd

```
library IEEE;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;

package fft_pkg is
      type ioarray is array (integer range <>) of std_logic_vector(255
downto 0);
      function log2(A: integer) return integer;
end;

package body fft_pkg is
      function log2(A: integer) return integer is
      begin
      for I in 1 to 30 loop  -- Works for up to 32 bit integers
                  if(2**I > A) then return(I-1);
                  end if;
      end loop;
    return(30);
      end;
end;
```

Appendix C: MATLAB Code

twiddlegen_rc.m

```
function twiddlegen_rc(N,tbits)
%   twiddlegen_rc(N,tbits)
%       This function generates all the roms needed for an
%       FFT of N points.  Twiddle factors are tbits wide.
%
%       This program uses:
%       romgen_rc.m
%           |-frac2bin.m
%           |-writebin.m
%

numpoints=N;
rnum=1;
while numpoints>4
    romgen_rc(numpoints,N,tbits,rnum);
    rnum=rnum+1;
    numpoints=numpoints/4;
end
```

romgen_rc.m

```
function romgen_rc(rp,fp,tbits,rnum)
%   romgen_rc(rp,fp,tbits)
%       rp= number of points in this rom
%       fp= total number of points in the FFT.
%       tbits=width of the twiddle factor
%       rnum=rom number
%
%       This function creates the vhdl ROM file used to store the
twiddle factors.
%       The resulting file is named rom<rnum>.vhdl, where <rnum> is the
value specified in rp.
%       For example: romgen(16,64,10,1) would create a file called
rom1.vhdl
%
%       This program uses:
%           frac2bin.m
%           writebin.m

%opening file for writing
fname=sprintf('rom%d.vhd',rnum);
fprintf('creating file %s\n',fname);
fid=fopen(fname,'w');
%writing beginning stuff to the file
aw=log2(rp);
fprintf(fid,'-- Rom file for twiddle factors \n');
fprintf(fid,'-- %s',fname);
fprintf(fid,' contains %d points of %d width \n',rp,tbits);
fprintf(fid,'-- for a %d point fft.\n\n',fp);
```

```
fprintf(fid,'LIBRARY ieee;\nUSE ieee.std_logic_1164.ALL;\nUSE
ieee.std_logic_arith.ALL;\n');
fprintf(fid,'\n\nENTITY rom%d IS\n    GENERIC(\n',rnum);
fprintf(fid,'        data_width : integer :=%d;\n',tbits);
fprintf(fid,'        address_width : integer :=%d\n',aw);
fprintf(fid,'    );\n    PORT(\n');
fprintf(fid,'        address :in std_logic_vector (%d downto 0);\n',aw-
1);
fprintf(fid,'        datar : OUT std_logic_vector (data_width-1 DOWNTO
0);\n');
fprintf(fid,'        datai : OUT std_logic_vector (data_width-1 DOWNTO
0)\n    );\n');
fprintf(fid,'end rom%d;\n',rnum);
%begin writing architecture
fprintf(fid,'ARCHITECTURE behavior OF rom%d IS\n\n BEGIN\n\n',rnum);
fprintf(fid,'process(address)\nbegin\n    case address is\n');
ma=fp/rp*[2 1 3];
address=0;
for m=1:3
    for n=0:((rp/4)-1)
%        fprintf('%d %d %d %d %d',n,m,ma(m),rp,fp);
        expval=exp(-2*pi*j*n*ma(m)/fp);
        rscld=round(real(expval)*(2^(tbits-1)-1));
        iscld=round(imag(expval)*(2^(tbits-1)-1));
        bitvecr=frac2bin(rscld,tbits,0);
        bitveci=frac2bin(iscld,tbits,0);
        addrvec=dec2bin(address,aw);
        fprintf(fid,'        when "%s" => datar <= "',addrvec);
        writebin(fid,bitvecr);
        fprintf(fid,'";datai <= "');
        writebin(fid,bitveci);
        fprintf(fid,'"; --%d\n',n*ma(m));
        address=address+1;
    end
end
%filling out the remaining zeros
bitvecr=frac2bin((2^(tbits-1)-1),tbits,0);
bitveci=frac2bin(0,tbits,0);
for n=0:(rp/4-1)
    addrvec=dec2bin(address,aw);
    fprintf(fid,'        when "%s" => datar <= "',addrvec);
    writebin(fid,bitvecr);
    fprintf(fid,'";datai <= "');
    writebin(fid,bitveci);
    fprintf(fid,'"; --0\n');
    address=address+1;
end

fprintf(fid,'        when others => for i in data_width-1 downto 0
loop\n');
fprintf(fid,'            datar(i)<=''0'';datai(i)<=''0'';end loop;\n');
fprintf(fid,'    end case;\n\n');
fprintf(fid,'end process;\nEND behavior;\n');
fclose(fid);
```

frac2bin.m

```
function [output]=frac2bin(a,ibits,fbits);
%  [output]=frac2bin(a,ibits, fbits);
%           a=number to be converted
%           ibits=integer part bits, must be an integer >0
%           fbits=number of fractional bits, must be an integer>=0
%       This returns an array [output] that is the input number in twos
complement form
%       [output] is ibits+fbits long
%    **NOTE**   This is not idiot proof, and will cause problems if the
number is too big
%               for the number of bits specified-1.

if (a>=0)
    bitsign=1;
    number=a;
else
    bitsign=-1;
    number=(a*-1);
end
ipart=number-rem(number,1);
fpart=rem(number,1);
unout(ibits+fbits)=0;
signedzero=1;

if ibits~=1
    for k=(ibits-1):-1:1,
        if ipart>=2^(k-1)
            unout(k+fbits)=1;
            ipart=ipart-2^(k-1);
            signedzero=0;
        end
    end
end

for k=fbits:-1:1,
    fpart=fpart*2;
    if fpart>=1
        unout(k)=1;
        fpart=fpart-1;
        signedzero=0;
    end
end

if (bitsign==-1) & (signedzero==0)
    testbit=1;
    for k=1:(ibits+fbits-1),
        if ((testbit==1) & (unout(k)==1))
            testbit=0;
        elseif ((testbit==0) & (unout(k)==0))
            unout(k)=1;
        elseif ((testbit==0) & (unout(k)==1))
            unout(k)=0;
```

```
        end
    end
    unout(ibits+fbits)=1;
end
output=unout;
```

bin2frac.m

```
function [output]=bin2frac(a,ibits,fbits);
%  [output]=frac2bin(a,ibits, fbits);
%           a=array to be converted to real numbers
%           ibits=integer part bits, must be an integer >0
%           fbits=number of fractional bits, must be an integer>=0
%       This returns a real number in [output]
%    **NOTE**   This is not idiot proof, and will cause problems if the
number is bigger
%               than bits specified.  It also assumes 2's complement
form

%code to convert back from signed to sign/magnitude
unin=a;
bitsign=1;
if unin(ibits+fbits)==1
    bitsign=-1;
    testbit=1;
    for k=1:(ibits+fbits),
        if ((testbit==1) & (unin(k)==1))
            testbit=0;
        elseif ((testbit==0) & (unin(k)==0))
            unin(k)=1;
        elseif ((testbit==0) & (unin(k)==1))
            unin(k)=0;
        end
    end
end
temp=0;
bitvalue=pow2(ibits-2);
for k=(ibits+fbits-1):-1:1,
    temp=temp+unin(k)*bitvalue;
    bitvalue=bitvalue/2;
end
output=temp*bitsign;
```

writebin.m

```
function writebin(fid,a);
%   writebin(fid,a)
%       fid - file id obtained from fopen
%       a - array to be written to to the file
%       No return arguements
for k=(size(a,2)):-1:1
    fprintf(fid,'%1.1d',a(k));
end
```

testveccon.m

```
function testveccon(N,bw,vecfile)
%testveccon(N,bw,vecfile)
%   This function generates a concatenated single line input for the
FFT.
%   It creates a single line input of the following format
%   resetn  load_enable  xin_r  xin_i
%
%   N is the number of points in the fft
%   bw is the bit width of the input
%   vecfile is the vector file name
%
%   Vectors will be some initial set up, then a sinc wave input,
%   followed by two square wave inputs.  This file does not generate
the
%   MATLAB calculated results, only the input waves.
%   Input waves are 1 bit of integer, bw-1 bits of fraction
%
%   This file uses:
%       frac2bin.m
%       writebin.m
%

 dutycycle=0.125;
 datara=1:N;
 datarb=1:N;
datara=sinc((datara-N/2)/2);
for k=1:N
    if k>(N*dutycycle)   %square wave input
         datarb(k)=0;
    else
         datarb(k)=1;
    end
end;

%dataia=datarb;
%dataib=datara;
%
%datara=(sin(2*pi*datara*12/64)+cos(2*pi*datara*2/64))/2;
dataia=0;
dataib=0;
datara=datara*(2^bw-1)/(2^bw);
datarb=datarb*(2^bw-1)/(2^bw);
fin=fopen(vecfile,'w');
writebin(fin,frac2bin(0,1,bw*2+1));
fprintf(fin,'\n1');
writebin(fin,frac2bin(0,1,bw*2));
for k=1:N
    fprintf(fin,'\n11');
    writebin(fin,frac2bin(datara(k),1,bw-1));
    writebin(fin,frac2bin(dataia(1),1,bw-1));
%    fprintf(fin,'\n');
end
```

```
for k=1:N
    fprintf(fin,'\n11');
    writebin(fin,frac2bin(datarb(k),1,bw-1));
    writebin(fin,frac2bin(dataib(1),1,bw-1));
%    fprintf(fin,'\n');
end

for k=1:N
    fprintf(fin,'\n11');
    writebin(fin,frac2bin(datarb(k),1,bw-1));
    writebin(fin,frac2bin(dataib(1),1,bw-1));
%   fprintf(fin,'\n');
end

fclose('all');
```

str2frac.m

```
function [output]=str2frac(a,ibits,fbits);
%  [output]=str2frac(a,ibits, fbits);
%           a= string of 1's and 0's to be converted to real numbers
%           ibits=integer part bits, must be an integer >0
%           fbits=number of fractional bits, must be an integer>=0
%       This returns a real number in [output]
%    **NOTE**   This is not idiot proof, and will cause problems if the
number is bigger
%               than bits specified.  It also assumes 2's complement
form

%code to convert back from signed to sign/magnitude
bitsign=1;
if a(1)=='1'
    bitsign=-1;
    testbit=1;
    for k=(ibits+fbits):-1:1,
        if ((testbit==1) & (a(k)=='1'))
            testbit=0;
            unin(k)=1;
        elseif ((testbit==1) & (a(k)=='0'))
            unin(k)=0;
        elseif ((testbit==0) & (a(k)=='0'))
            unin(k)=1;
        elseif ((testbit==0) & (a(k)=='1'))
            unin(k)=0;
        end
    end
else
    for k=1:(ibits+fbits),
        if a(k)=='1'
            unin(k)=1;
        end
        if a(k)=='0'
            unin(k)=0;
```

```
        end
    end
end
temp=0;
bitvalue=pow2(0-fbits);
for k=(ibits+fbits):-1:1,
    temp=temp+unin(k)*bitvalue;
    bitvalue=bitvalue*2;
end
output=temp*bitsign;
```

plotdata.m

```
function plotdata(infile,outfile,ibin,fbin,add_g,mult_g,N)
%  plotdata(infile,outfile,ibin,fbin,add_g,mult_g,N)
%   Reads in from infile (test vectors sent into fft_filed_tb
%   and outfile (output of fft_filed_tb, usually data.out)
%
%   ibin is integer bits in the input.
%   fbin is fractional bits in the input.
%   add_g is the adder growth.
%   mult_g is the multiplier growth.
%   N is the number of points in the FFT.
%
%   This program uses:
%       str2frac.m
%
clf
fout=fopen(outfile);
ibout=ibin+add_g*log2(N);
if mult_g>0
    ibout=ibout+floor((log2(N)-1)/2);
end;
fbout=fbin+(floor((log2(N)-1)/2)*(mult_g-1));
dp=ibout+fbout;
dp2=2*dp;
A=(fscanf(fout,'%s',[dp2,inf]))';
[nr,nc]=size(A);
hold on;
offset=N+2;
%getting first outputs
if (N+offset)<=nr
    for k=offset+1:offset+N
        out1r(k-offset)=str2frac(A(k,1:dp),ibout,fbout);
        out1i(k-offset)=str2frac(A(k,dp+1:dp2),ibout,fbout);
    end
      out1r=out1r';
      out1i=out1i';
      %Bit reversing and recombining outputs
      for k=0:(N-1)
        bitorder=dec2bin(k,log2(N));
        for m=1:(size(bitorder,2)/2)
            temp=bitorder(m);
```

```
            bitorder(m)=bitorder(size(bitorder,2)+1-m);
            bitorder(size(bitorder,2)+1-m)=temp;
        end
        bitrevpos=bin2dec(bitorder);
        adata(bitrevpos+1)=out1r(k+1)+i*out1i(k+1);
    end
    subplot(2,2,1),plot(fftshift(abs(adata)),'k'),axis tight,
    title('Magnitude of FFT of Sinc Wave Input'),hold on;
    subplot(2,2,2),plot(fftshift(angle(adata)),'k'),axis tight,
    title('Angle of FFT of Sinc Wave Input'),hold on;
end
%getting second outputs
offset=offset+N;
if (N+offset)<=nr
    for k=offset+1:offset+N
        out1r(k-offset)=str2frac(A(k,1:dp),ibout,fbout);
        out1i(k-offset)=str2frac(A(k,dp+1:dp2),ibout,fbout);
    end
    out1r=out1r';
      out1i=out1i';
      %Bit reversing and recombining outputs
      for k=0:(N-1)
        bitorder=dec2bin(k,log2(N));
        for m=1:(size(bitorder,2)/2)
            temp=bitorder(m);
            bitorder(m)=bitorder(size(bitorder,2)+1-m);
            bitorder(size(bitorder,2)+1-m)=temp;
        end
        bitrevpos=bin2dec(bitorder);
        bdata(bitrevpos+1)=out1r(k+1)+i*out1i(k+1);
      end
      subplot(2,2,3),plot(fftshift(real(bdata)),'k'),axis tight,
    title('Real part of FFT of Square wave input'),hold on;
      subplot(2,2,4),plot(fftshift(imag(bdata)),'k'),axis tight,
    title('Imaginary part of FFT of Square wave input'),hold on;
end


%getting input waves from testvec and outputting them on same plots
fin=fopen(infile);
dp=ibin+fbin;
dp2=2*dp+2;
dp=dp+2;
A=(fscanf(fin,'%s',[dp2,inf]))';
[nr,nc]=size(A);

offset=2;
%getting first inputs
if (N+offset)<=nr
    for k=offset+1:offset+N
        outdata(k-offset)=str2frac(A(k,3:dp),ibin,fbin) +
i*str2frac(A(k,dp+1:dp2),ibin,fbin);
    end
    afftdata=fft(outdata);
```

```
    subplot(2,2,1),plot(fftshift(abs(afftdata)));
    subplot(2,2,2),plot(fftshift(angle(afftdata)));
end

%getting second inputs
offset=offset+N;
if (N+offset)<=nr
    for k=offset+1:offset+N
        outdata(k-offset)=str2frac(A(k,3:dp),ibin,fbin) +
i*str2frac(A(k,dp+1:dp2),ibin,fbin);
    end
    bfftdata=fft(outdata);
    subplot(2,2,3),plot(fftshift(real(bfftdata)));
    subplot(2,2,4),plot(fftshift(imag(bfftdata)));
end


fclose('all');
```

Vita

Adam R. Miller was born in Hinsdale, Illinois, on March 24th 1978. He moved several times to places in Indiana, Kansas, Texas, and finally to Crystal Lake, Illinois. There he attended Crystal Lake South High School, and finished in May of 1996. After graduating from high school, his family moved to Tennessee, and he began attending the Knoxville campus of the University of Tennessee. It was at the university that he met his wife, Lara Stembridge of Memphis, Tennessee. In December of 2000 they were married in Memphis.

In May of 2000, he completed the Bachelor of Science degree in Electrical Engineering. He immediately entered the graduate program at UT, and began working as a Graduate Teaching Assistant. He assisted professors in assembly language and digital design classes until May of 2002, when he began working as a Graduate Research Assistant under Dr. Donald Bouldin. He has completed all the requirements for the Master of Science degree in Electrical Engineering and will be awarded that degree in August of 2003.