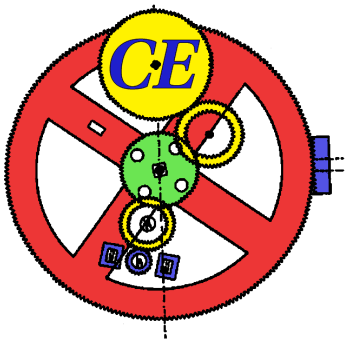

Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2009

MSc THESIS

A Custom FFT Hardware Accelerator for Wave Fields Synthesis

Reza Sadegh Azad

CE-MS-2009-01

Abstract

Wave Field Synthesis (WFS) is a sound reproduction technique using loudspeaker arrays that redefines the limits set by conventional techniques. The aim of this technique is to reproduce the true physical attributes of a given sound field over an extended area of the listening room. So far, this technology is only implemented on standard PCs. Besides, this concept requires to be performed in real-time systems and the question is how we can map such an application on an embedded system and meet the requirements to perform this concept as a real-time application. The aim of this thesis is to investigate the application of this concept for embedded processing devices and specially investigate how to optimize the time consuming kernels in order to make it suitable for real-time computation. Field-Programmable Gate Arrays (FPGAs) are chosen for this purpose as the target for embedded processing. Our approach is divided into a number of steps. Profiling is the first step to identify the time consuming kernels. During this stage we have profiled the application to assess the time consuming kernels. We found out that Fast Fourier Transform(FFT) is the most time consuming kernel of the application. The next step was to convert the kernel from a floating-point into fixed-point. Afterwards, we consider three different options for implementing a FFT as a reconfigurable stand alone processor on a FPGA. The result from this stage led to design of a custom 1024 points Radix-4 FFT unit. Finally, the design was simulated, synthesized and implemented on XC2VP30 Virtex 2Pro. At the simulation stage we could conclude that our FFT core was 8X faster then the software version, performed on Intel Core2 2.2 GHZ. We have performed the same approach for FFT core including MOLEN interface and found out that this unit was also 5X faster then the software version. Moreover, during systhesis stage we have achieved a Maximum Clock Frequency equal to 125 MHz. During the implementation stage the clock period was set to 10 ns, which led to having a successfull implementation of the custom FFT core. The results from fixed-point arithmetic were compared to the floating-point counterpart, which led to having a small difference of $2.8*10^{-8}$ for the real part and $1.1*10^{-5}$ for the imaginary part. Furthermore, our FFT design is compared to Xilinx core in term of accuracy, based on the results we could conclude that Xilinx core adds almost 5% to 8% errors to the calculated numbers. Finally, we have calculated the overall speedup for the application based on our custom FFT design, which led to achive 27% of speedup for the entire application.

TU Delft
Delft University of Technology

Faculty of Electrical Engineering, Mathematics and Computer Science

A Custom FFT Hardware Accelerator for Wave Fields Synthesis

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Reza Sadegh Azad
born in Tehran, Iran,

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

A Custom FFT Hardware Accelerator for Wave Fields Synthesis

by Reza Sadegh Azad

Abstract

Wave Field Synthesis (WFS) is a sound reproduction technique using loudspeaker arrays that redefines the limits set by conventional techniques. The aim of this technique is to reproduce the true physical attributes of a given sound field over an extended area of the listening room. So far, this technology is only implemented on standard PCs. Besides, this concept requires to be performed in real-time systems and the question is how we can map such an application on an embedded system and meet the requirements to perform this concept as a real-time application. The aim of this thesis is to investigate the application of this concept for embedded processing devices and specially investigate how to optimize the time consuming kernels in order to make it suitable for real-time computation. Field-Programmable Gate Arrays (FPGAs) are chosen for this purpose as the target for embedded processing. Our approach is divided into a number of steps. Profiling is the first step to identify the time consuming kernels. During this stage we have profiled the application to assess the time consuming kernels. We found out that Fast Fourier Transform(FFT) is the most time consuming kernel of the application. The next step was to convert the kernel from a floating-point into fixed-point. Afterwards, we consider three different options for implementing a FFT as a reconfigurable stand alone processor on a FPGA. The result from this stage led to design of a custom 1024 points Radix-4 FFT unit. Finally, the design was simulated, synthesized and implemented on XC2VP30 Virtex 2Pro. At the simulation stage we could conclude that our FFT core was 8X faster then the software version, performed on Intel Core2 2.2 GHZ. We have performed the same approach for FFT core including MOLEN interface and found out that this unit was also 5X faster then the software version. Moreover, during systhesis stage we have achieved a Maximum Clock Frequency equal to 125 MHz. During the implementation stage the clock period was set to 10 ns, which led to having a successfull implementation of the custom FFT core. The results from fixed-point arithmetic were compared to the floating-point counterpart, which led to having a small difference of $2.8*10^{-8}$ for the real part and $1.1*10^{-5}$ for the imaginary part. Furthermore, our FFT design is compared to Xilinx core in term of accuracy, based on the results we could conclude that Xilinx core adds almost 5% to 8% errors to the calculated numbers. Finally, we have calculated the overall speedup for the application based on our custom FFT design, which led to achive 27% of speedup for the entire application.

Laboratory : Computer Engineering
Codenumber : CE-MS-2009-01

Committee Members :

Advisor:	Georgi Kuzmanov, CE, TU Delft
Advisor:	Dimitris Theodoropoulos, CE, TU Delft

Chairperson:	Koen Bertels, CE, TU Delft
Member:	Diemer de Vries, TU Delft

In the memory of my grandfather and grandmother.

Contents

List of Figures viii

List of Tables ix

Acknowledgements xi

1 Introduction 1
- 1.1 Related Work . . . 1
- 1.2 Thesis Objectives . . . 2
- 1.3 Thesis Organization . . . 3

2 Wave Field Synthesis 5
- 2.1 Introduction . . . 5
- 2.2 WFS . . . 5
- 2.3 Data Separation . . . 6
- 2.4 Frequency-domain convolution . . . 7
- 2.5 Overlap-add and overlap-save . . . 7
  - 2.5.1 Overlap-save . . . 7
  - 2.5.2 Overlap-add . . . 8
- 2.6 Real-time convolution of long signals . . . 9

3 Profiling 11
- 3.1 Introduction . . . 11
- 3.2 Profiling Result . . . 11
- 3.3 Computation Units . . . 13

4 Fast Fourier Transform 15
- 4.1 Introduction . . . 15
- 4.2 Fast Fourier Transform Algorithms . . . 15
- 4.3 Basic Concepts of FFT Algorithms . . . 15
  - 4.3.1 Divide-and-Conquer . . . 15
  - 4.3.2 Properties of the Coefficient(W_N^{nk}) . . . 16
- 4.4 Classification of FFT Algorithms . . . 18
  - 4.4.1 Fixed-Radix FFT Algorithms . . . 18
  - 4.4.2 Radix-2 FFT Algorithm . . . 18
  - 4.4.3 Radix-4 FFT Algorithm . . . 22
  - 4.4.4 Radix-2^2 DIF FFT Algorithm . . . 26
  - 4.4.5 Summery . . . 29
- 4.5 FFT HARDWARE ARCHITECTURES . . . 30
  - 4.5.1 Pipeline-Based FFT Architecture . . . 30
  - 4.5.2 Memory-Based FFT Architecture . . . 34
  - 4.5.3 Summary . . . 35

5 Fixed-Point Representation **37**
5.1 Introduction . . . 37
5.2 Fixed-Point Range - Integer Portion . . . 37
5.3 Fixed-Point Resolution - Fractional Portion . . . 38
5.4 Converting FFT unit From Floating-point into Fixed-point . . . 39
5.5 Modification of FFTld Unit . . . 44

6 FFT Hardware Considerations **47**
6.1 Introduction . . . 47
6.2 The Molen Processor . . . 47
6.3 Xilinx FFT Core . . . 48
6.3.1 Xilinx FFT Core Simulation . . . 53
6.3.2 Results of FFT core . . . 54
6.4 DWARV tool chain . . . 59

7 Custom FFT Design **61**
7.1 Introduction . . . 61
7.2 Implementation of the FFT processor . . . 61
7.3 Memory-Based FFT Architecture . . . 61
7.3.1 Block Diagram of the FFT Processor . . . 62
7.3.2 Input Unit . . . 63
7.4 Components of the FFT processor . . . 65
7.4.1 Data Address Generator Unit (DAG) . . . 65
7.4.2 FFT Write Back Address . . . 67
7.4.3 Butterfly Counter . . . 67
7.4.4 ROM Address Generator . . . 68
7.4.5 Butterfly Processing Element . . . 68
7.4.6 Processing Elements . . . 69
7.4.7 Controller Unit . . . 71
7.4.8 ROM . . . 71
7.4.9 RAM . . . 72
7.4.10 CCU Controller Unit . . . 73

8 Design Evaluation and Experimental Results **75**
8.1 Synthesis of the FFT Processor . . . 75
8.2 Simulation of the FFT Processor . . . 76
8.3 Related Work . . . 80
8.4 Performance evaluation . . . 81
8.5 Design comparison . . . 83

9 Conclusions **85**
9.1 Summary . . . 85
9.2 Objective Coverage . . . 86
9.3 Further work . . . 87

10 References **89**

List of Figures

2.1 Huygens principle for a spherical wavefront. 6
2.2 overlap-save process. Input signal X[n] is divided in three parts, each equal to L size blocks. Then the convolution is performed by using x[i] and y[i]. Finally the extra M-1 parts are discarded to construct the Y[n] signal 8
2.3 overlap-add process. Input signal X[n] is divided into three parts, each equals into L size blocks.Then the convolution is performed by using x[i] and y[i]. Finally the extra parts are summed up to construct the Y[n] signal 8
2.4 partitioned convolution process[2]. The input signal is broken into blocks and each block is Fourier transformed, convolved, inverse transformed an overlap-saved to the output signal. 9

3.1 Filtering Process. 13

4.1 DFT equation 15
4.2 The partial twiddle factor of an N-point DFT. 16
4.3 First stage of the radix-2 DIF FFT algorithm 20
4.4 Signal flow graph of a typical 2-point DFT 20
4.5 Radix-2 DIF FFT signal flow graph of 16-point 21
4.6 The basic butterfly for radix-2 DIF FFT algorithm. 21
4.7 Four diagram depicting bit-reversed sorting [13]. 22
4.8 The basic butterfly for radix-4 DIF FFT algorithm. 24
4.9 Radix-4 signal flow graph. 24
4.10 Radix-4 signal flow graph. 25
4.11 Four diagram depicting digit-reversed sorting. 26
4.12 The basic butterfly for radix-22 DIF FFT algorithm. 27
4.13 Four diagram depicting digit-reversed sorting. 28
4.14 Projection mapping of the radix-2 DIF signal flow graph. 31
4.15 Radix-2 Single-path Delay Feedback (N=16). 32
4.16 The first two stages of N-point radix-2 MDC structure. 32
4.17 The first two stages of N-point radix-2 MDC structure. 34

5.1 FFT Unit Overview 39
5.2 Fixed-Point FFT Butterfly in C 44
5.3 Radix-2 Fixed-Point FFT Butterfly 45

6.1 The MOLEN machine organization 48
6.2 Xilinx FFT Core 49
6.3 Activate the input signals 51
6.4 Read data from FFT Core 51
6.5 Unload data from FFT Core 52
6.6 Valid data count 52
6.7 write process 52
6.8 Xilinx FFT Core Simulation 53
6.9 Xilinx Real numbers(Precision). 54
6.10 Xilinx Real numbers(Accuracy). 55
6.11 Xilinx Real numbers(Relative Error). 55
6.12 Xilinx Imaginary numbers(Accuracy). 56
6.13 Xilinx Imaginary numbers(Precision). 57

6.14 Xilinx Imaginary numbers(Relative Error). 58

7.1 Top level description of the operation of the FFT processors. 62
7.2 General overview of Block diagram of the FFT processor. 63
7.3 Input-Unit of the FFT Processor. 64
7.4 The Core of the FFT Processor. 65
7.5 Block diagram of a data address generator. 67
7.6 Radix2^2 coefficient index generator. 69
7.7 Block diagram of the butterfly processing element. 69
7.8 *Radix*2^2 Processing element. 70
7.9 One of *Radix*2^2 butterfly processing element. 71
7.10 State Diagram for a FFT processor controller. 72
7.11 State Diagram for a MOLEN controller. 73

8.1 FFT Results of Real Numbers. 78
8.2 FFT Results of Imaginary Numbers. 79

List of Tables

3.1 Profiling Results 12

4.1 comparisons of FFT algorithms 29
4.2 The hardware utilization and requirement in different structure 33

5.1 coefficients 42

7.1 Data address for butterfly PE in direct processing order. 66

8.1 Synthesis results of FFT Core 75
8.2 Synthesis results of FFT Core including MOLEN Interface 76
8.3 Performance comparison with other fixed-point FFT processors [30] [31] [32][33] . 80
8.4 The hardware utilization and requirement for different FFT architecture 83

Acknowledgements

I would like to thank my advisors Georgi Kuzmanov and Dimitris Theodoropoulos for guiding me during my MSc thesis. I also want to thank all CE members and my family for their support during my education.

Reza Sadegh Azad
Delft, The Netherlands
December 12, 2009

1 Introduction

Throughout the history of computing, digital signal processing applications have pushed the limits of computing power, especially in terms of real-time computation. In order to solve the shortcomings, three ways of supporting processing requirements are available.
The first option would be to consider high performance microprocessors(μPs). However, even expensive μPs are not fast enough for many DSP applications. Besides, this approach usually requires a high power consumption(100W or more) and is not really convenient for embedded applications.
The second option would be to use application specific ICs (ASICs). The benefits of this approach would be to have a natural mechanism for large amount of parallelism and consumes less power than reconfigurable devices. However, the down side of this approach is that it is not suitable for general purpose processing and would require a long time of development. It is also infeasible for many embedded systems, since only high volume production makes it cost efficient.

The final option would be to consider the Reconfigurable Computing technique. This concept is a combination between μP and ASICs and offers a high performance and lower power consumption. It is also capable to offer a great parallelism due to hardware implementation and is especially cheaper for low volume production. It also offers a short design time, time to market is much less[34] and is specially a good candidate for General Purpose. The aim of this thesis is to investigate the number of time consuming kernels from a 3-D audio application using the Wave Fields Synthesis(acoustic algorithm) and to accelerate these computational intensive kernels. In order to meet this requirement our approach is to extend a general purpose processor with the hardware implementations of important kernel functions. This involves considering the Reconfigurable concept, which imply a great possibility for this design. Besides, it also offers us a great freedom to implement the time consuming kernels as a Core-processor next to a General Processor unit. The MOLEN polymorphic processor [6] developed at the Delft University of Technology is utilized during this thesis to accomplish this task.

1.1 Related Work

There are not many publications available regarding hardware implementations of the WFS. The reason for this is because this concept is still a new area for research. In addition, this concept is so far executed only on standard PCs and is not really considered with other technologies. However, there are two papers published so far, which dealing with mapping the WFS application onto FPGAs. We will give a brief introduction of these publications:

- In[5] Dimitris Theodoropoulos and his colleges have investigated the architectural perspective of the Wave Field Synthesis (WFS) 3D-audio algorithm mapped on three different platforms: a General Purpose Processor (GPP), a Graphics Processor Unit (GPU) and a Field Programmable Gate Array (FPGA). First, they have considered using contemporary GPUs, which consist of many multiprocessors to process data concurrently. They have also considered using FPGAs, because of huge level of parallelism, and reasonably high performance. Furthermore, they have implemented a reconfigurable and

scalable hardware accelerator for the WFS kernel, and map it onto Virtex4 FPGA and GeForce 8600GT GPU. Finally they have compared both architectural approaches against a baseline GPP implementation on a Pentium D at 3.4 GHz. They have concluded that in highly demanding WFS based audio systems, a low-cost GeForce 8600GT desktop GPU can achieve a speedup of up to 8x comparing to a modern Pentium D implementation. However the same approach on an FPGA-based WFS hardware accelerator leads to having a speedup of up 10x comparing to the Pentium D approach.

- In[4], the same Authors have presented a scalable organization comprising multiple rendering units (RUs), each of them independently processing audio samples. A hardware prototype of their proposal was implemented on a Virtex4FX60 FPGA operating at 200 MHz. According to their conclusion a single RU can achieve up to 7x WFS processing speedup compared to a software implementation running on a Pentium D at 3.4 GHz. Moreover, the power consumption according to Xilinx XPower is estimated to approximately 3 W.

1.2 Thesis Objectives

The aim of this thesis is to investigate how to implement a 3D-audio application using Wave Fields Synthesis algorithm onto an FPGA through a reconfigurable technique. The primary target is to determine the time consuming kernels in order to implement them into hardware to accelerate the WFS application. To accomplish this task, we have divided our approach into number of steps namely:

- The main objective of this thesis is to investigate how to speed up the WFS application by using the reconfigurable computing.
- in order to speed up the application, the application has to be profiled to identify the time consuming kernels.
- in order to speed up the identified kernels, the first step is to investigate how to convert the time consuming kernel from the floating-point into the fixed-point format.
- Several literature studies must be done in order to obtain an overview of available options for implementing the identified kernels into the hardware. Besides these options also has to be investigated in term of error numbers, since this is also an important requirement during this thesis.
- Based on these studies, a proper choice must be taken to implement the kernel into the hardware.
- Finally, the design has to be integrated into the MOLEN processor in order to be executed as a reconfigurable hardware accelerator of the MOLEN processor.

1.3 Thesis Organization

This thesis is organized as follows: in chapter 2 we give a brief introduction about Wave Field Synthesis. We also mention two different convolution techniques, which are used in Digital Signal Processing in order to deal with long data sequence in frequency-domain. We will also describe how to deal with real-time convolution of long data sequences. In chapter 3 we discuss about the results of profiling. Profiling was necessary in order to obtain a clear identification about the time consuming kernels. In chapter 4 we have described the background of the identified time consuming kernel. Several algorithms are described, which can be applied for the kernel. We have also explained about the hardware architectures, which should be considered during the design of a custom core. In chapter 5 the identified time consuming kernels are converted from floating-point into fixed-point format. This was necessary in order to speed up the computation, since floating point requires more cycles during its execution in contrast to fixed-point approach. In chapter 6 three different options are considered to implement a fixed-point FFT processor. In chapter 7 a custom 1024 points Radix-4 FFT processor is proposed and implemented in VHDL. In chapter 8, the results of custom FFT processor and FFT software are discussed and evaluated. Finally, in chapter 9 we have concluded about our thesis and discussed about the objectives and the future work.

2 Wave Field Synthesis

2.1 Introduction

State-of-the-art systems for the reproduction of 3D-audio suffer from a serious problem: The spatial properties of the reproduced sound can only be perceived correctly in a small part of the listening area, the so-called sweet spot. This restriction occurs because conventional reproduction of 3D audio is based on psychoacoustics, i.e. mainly intensity panning techniques. A solution to this problem calls for a new reproduction technique which allows the synthesis of physically correct wave fields of three-dimensional acoustic scenes. Furthermore, it should result in a large listening area which is not restricted to a particular sweet spot.

A number of different approaches have been suggested. They can be categorized into advanced panning techniques [35], ambisonics [36] and wave field synthesis [1]. Advanced panning techniques aim is to enlarge the sweet spot well known from two-channel stereo or 5-channel surround sound systems. Ambisonics is a technique, for recording, studio processing and reproduction of the complete sound field experienced during the original performance. Ambisonic technology does this by decomposing the directionality of the sound field into spherical harmonic components, termed W, X, Y and Z. The Ambisonic approach is to use all speakers to cooperatively recreate these directional components.

This chapter presents the third approach from above, wave field synthesis (WFS). It is a technique for reproducing the acoustics of large recording rooms in smaller sized listening rooms. WFS typically using more loudspeakers than audio channels. Contrary to conventional spatial audio reproduction techniques like two-channel stereo or 5-channel surround, there exists no fixed mapping of the audio channels into the reproduction loudspeakers. Instead, a representation of the three-dimensional acoustic scene is used that can be reproduced by various WFS systems utilizing different loudspeaker setups.

2.2 WFS

WFS is based on Huygens principle. It states that the propagation of a wave through a medium can be formulated by adding the contributions of all of the secondary sources positioned along a wave front [1].This is demonstrated in figure 1.1 for a spherical wavefront.

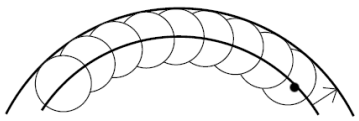

Figure 2.1: Huygens principle for a spherical wavefront.

This principle can be used to synthesize acoustic wave fronts of arbitrary shape. By placing the loudspeakers on an arbitrary fixed curve and by weighting and delaying the driving signals, an acoustic wave front can be synthesized with a loudspeaker array. For more information about WFS theory, the reader can refer to the next references [1-37].

There are two different approaches for applying WFS in practice. The first way is to record the direct sound as well as the reverberation of the room. This must be done by using separate channels during a recording session. Spot microphones together with special room configuration are needed during this step. The second way is to record the direct sound during a session. The acoustics environment, which are the impulse responses have to be measured individually during a session for each location of the source. This information can be applied during the playback by convolving the direct sound with impulse responses. The result from previous step can be reproduced by the loudspeaker array using plane waves. This method is considered in this thesis. The WFS method has some drawbacks as well as advantages regarding to other conventional techniques. The main advantages of the WFS compared to other techniques are:

- WFS make it possible to reproduce the dry recorded source signal in several acoustic environments with arbitrary listener and source positions.
- The biggest advantage is that this technique is able to constract a proper 3D sound, which leads to generating a sweet area instead of the sweet spot.

There are also some other shortcomings, which are involved with this technique:

- To be able to apply WFS during a playback, we have to use a large array of speakers. This leads to having a costly construction.
- The other disadvantage of WFS is the need to use convolution technique to reproduce an audio playback, which can be a very intensive computational task for the underlying system.

2.3 Data Separation

As it was discussed in the previous section, WFS divides the data into two kinds of information. This information consists of dry signal and acoustic environment. The dry signal is the information from an individual source or group of sources. This signal is totally free of any extra information and contains only a dry signal (e.x. singer voice). However, acoustic environment is the set of impulse responses and is used as the room configuration during the playback.

The impulse response is the response of a system to an impulse and can be defined as a Dirac pulse. It exposes how a system reacts to such an impulse. The impulsive audio signal usually is used to measure the response of a room. This information can be convoluted during an audio playback with a dry signal to obtain a desired signal. In general, an impulse response can be divided in three parts:

- Direct sound
- Early reflections
- Reverberation tail

The direct sound is the impulse that is directly transferred to the listener. Early reflection is caused by reflecting sound via floor, wall and ceiling. The final case is actually a relation between the first and the second case and is responsible for the last audio effect before reaching the loudspeakers.

2.4 Frequency-domain convolution

Convolution in real-world systems is mostly done in the frequency domain, especially when an impulse response becomes long. Usually, there is a requirement to perform a real-time convolution when working with audio-acoustic variable systems. However, traditional time-domain techniques will suffer from a heavy computation which is caused by an enormous number of multiplications at this stage.

To overcome this problem, overlap-add and overlap-save convolution is used. This approach is a good candidate to solve the mentioned problem. It also introduces a new set of problems, when using a block-based algorithm with an incoming stream of data. To successfully apply this approach, the data sequence must be divided into subsequences. This step is similar to divide and conquer approach, which is a familiar solution for most software problems. The calculation is divided into number of blocks. This is done by applying the overlap-save algorithm as follows: Each block is shifted, doubled in length, convolved and summed up. The overlap-add technique is less efficient since more calculations are required. However this technique allows to have much lower latency due to one partition of the impulse response instead of the total response.

2.5 Overlap-add and overlap-save

In time-domain, a convolution can be considered as a weighted summed integral of the form:

$$y(t) = x(t) * h(t) = \int_{-\infty}^{+\infty} x(\tau)h(t-\tau)d\tau$$

where x(t) is the input signal, h(t) is a transfer function of some system that the input signal passes and y(t) is the output signal. Important properties of the convolution are (1) linearity, (2) commutativity and (3) the fact that convolution in time-domain corresponds to a frequency-domain product, i.e., element-wise multiplication of the Fourier spectra:

$$Y(j\omega) = X(j\omega)H(j\omega) = F\{x(t) * h(t)\}$$

where upper-case are complex Fourier spectra and F{.} is the Fourier transform.

2.5.1 Overlap-save

This method starts by dividing the input signal into frames of length L and fills only the first frame with zeros. After the convolution of a block, the end of the data of this block will be polluted with wrap-around effects and thus be thrown away. This method will cause a delay of L (the size of the impulse response). This process is shown in figure 2.2:

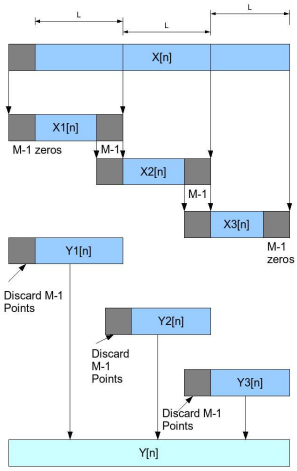

Figure 2.2: overlap-save process. Input signal X[n] is divided in three parts, each equal to L size blocks. Then the convolution is performed by using x[i] and y[i]. Finally the extra M-1 parts are discarded to construct the Y[n] signal

2.5.2 Overlap-add

This method is used to work with data in blocks. This means that the input signal is segmented into frame blocks, then each block is zero padded at both ends and then convolved. Results of subsequent frames are added, including the overlapping regions formed by the zero-padded parts. The next figure shows this process:

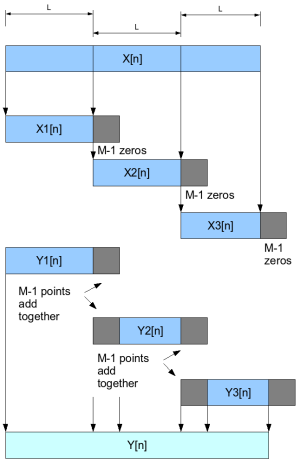

Figure 2.3: overlap-add process. Input signal X[n] is divided into three parts, each equals into L size blocks.Then the convolution is performed by using x[i] and y[i]. Finally the extra parts are summed up to construct the Y[n] signal

2.6 Real-time convolution of long signals

All these techniques discussed so far are partly done in the time domain and partly in the frequency domain. The input impulse response h(n) is divided into a number of equally sized blocks. Furthermore, each block needs to be convolved according to an overlap-save method as it was explained in the previous section. Afterwards, the resulting sub blocks are Fourier Transformed and multiplied with a block of the input signal. At the end the sub-blocks are delayed to their original position and summed. An overview of this process is shown in figure 2.4. Because of its flexibility, this technique is a good candidate for the software implementation. By

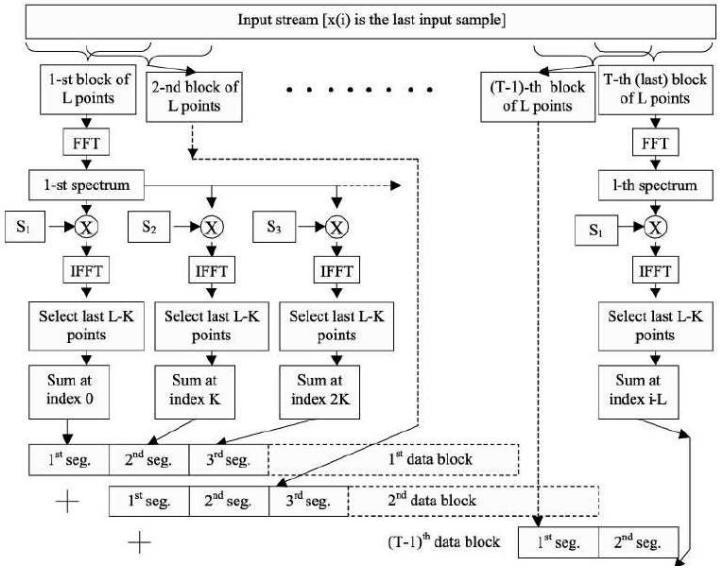

Figure 2.4: partitioned convolution process[2]. The input signal is broken into blocks and each block is Fourier transformed, convolved, inverse transformed an overlap-saved to the output signal.

partitioning the whole process into number of stages, each stage can individually be implemented in software. This can be seen from figure 2.4, which shows each process for this technique.

3 Profiling

3.1 Introduction

Profiling allows identifying, which are the most computationally intensive functions of an application. Furthermore, it shows where the application spent its time and which functions are called by other functions while they were executing. This information can be used to find out which pieces of the application are time consuming that might be candidates for rewriting or a hardware implementation to make the program execute faster. It can also give an overview of which functions are being called more or less than expected. In this section we will discuss the results of this step, which was performed as the first step to analyze the WFS application.

3.2 Profiling Result

To accomplish this task we have applied the Gprof program, which is available under Linux operating system. Because the application is compiled by using GNU-compiler, Gprof can easily be included during the compilation process. This is executed by adding the necessary flags to GCC compiler to allow the compiler to generate a file, which covers all necessary information about the application kernels. Since, the application is a real time application, a shell script is written to cause the application to be executed in a number of times. We have chosen to run the application for 294 times, which should be a good starting point to have an indication of time consuming kernels. There are several forms of output files available, which can be divided as follows:

- Flat profile
- Call graph
- Annotated source

The flat profile is selected, because this option shows simply how much time the program spent in each function and how many times that function was called. It also shows the total amount of time the program spent executing each function. Table 3.1 shows the WFS profiling results. The functions are sorted first by decreasing run-time spent in them, then by decreasing a number of calls, then alphabetically by name. Just before the column headers, a statement appears indicating how much time each sample counted as. This sampling period estimates the margin of error in each of the time figures. A time figure that is not much larger than this is not reliable. In this file, each sample counted as 0.01 seconds, suggesting a 100 Hz sampling rate. The program's total execution time was 294.26 seconds, as indicated by the cumulative seconds field. Since each sample counted for 0.01 seconds, this means only 29426 samples were taken during the run. The next table 3.1 shows the results of this process:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	self calls	total ms/call	ms/call	name
36.50	107.42	107.42	396500	0.27	0.27	fft1d
25.38	182.12	74.70	173000	0.43	0.47	DelayLine_processChunk
8.51	207.16	25.04	406016000	0.00	0.00	bitrev
7.59	229.49	22.33	500	44.66	588.43	waveprop
7.38	251.21	21.72	200000	0.11	0.56	DelayLine_processFrame
3.03	260.12	8.91	396500	0.02	0.09	perm
2.66	267.95	7.83	2205500	0.00	0.00	zeroRealVec
2.18	274.36	6.41	13824500	0.00	0.00	cycinc
2.12	280.61	6.25	200000	0.03	0.80	Filter_process
1.51	285.05	4.43	204800000	0.00	0.00	cmult
1.03	288.09	3.04	204800000	0.00	0.00	cadd
0.92	290.79	2.69	201000	0.01	0.01	zeroCplxVec
0.37	291.88	1.09	200000	0.01	0.01	c2r
0.35	292.92	1.04	196000	0.01	0.01	r2c
0.23	293.60	0.68	12924500	0.00	0.00	DelayLine_updateChunk
0.05	293.74	0.14	1000	0.14	0.14	ffw
0.04	293.86	0.12	500	0.25	0.25	ldint
0.04	293.98	0.12	200000	0.00	0.00	AudioIo_setFrames
0.03	294.07	0.09	195500	0.00	0.00	controlrun
0.03	294.14	0.07				csub
0.02	294.20	0.06	195500	0.00	0.00	AudioIo_getFrames
0.01	294.23	0.03	200000	0.00	0.35	Filter_process_pre
0.01	294.26	0.03	500	0.06	542.88	audiorun

Table 3.1: Profiling Results

At this stage we had a clear indication of the time consuming kernels. The next step was to find out a relation between these kernels according to table 3.1. Since this figure depicts the whole computation process of WFS structure, we have to reorder these kernels in way that they correspond to units of table 3.1. This was necessary because our aim was to design an independent unit in hardware and not just speedup an individual kernel. The next section will discuss this part in more details.

3.3 Computation Units

In this section we will discuss the time consuming kernels according to our profiling result. From the profiling result, we can divide the whole application into two parts as illustrated in figure 3.1. Each of these parts can be considered as the most time consuming part of the application. These parts require a huge number of computations. The first unit can be considered as a FIR filter unit. This unit starts by taking the samples, which are originated from sampling unit. The FFT transform is used to convert the time domain samples into frequency domain. After this stage, the next step is to filter the audio samples by multiplying them with the desired coefficients. Finally each block is transformed back into the time domain again for a further processing. The whole process can be defined as follows:

- The dry signal, which contains the original audio is divided into a number of blocks. Hereafter each block is transformed from Time domain to frequency domain by using Fast Fourier Transform.
- In this stage every transformed block is multiplied with FIR filter coefficients. Filter coefficients are actually the room configurations and involve all necessary information about the area. The information is applied in this stage to perform convolution based on dry signal and room information.
- IFFT is considered as the last stage of filtering process. It is responsible to perform an Inverse Fast Fourier Transform. In this stage each block of the filtered audio signals are converted from frequency domain into the Time domain.

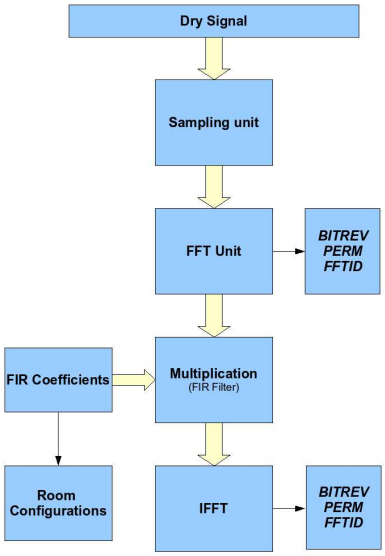

Figure 3.1: Filtering Process.

As it is shown in the previous figure, three operations are involved in these stages. Each of these operations is a part of FFT/IFFT transform. They appear also in profiling table and belong to the most time consuming operations of WFS application. Also other operations (DelayLine_processChunk, DelayLine_processFrame and DelayLine_updateChunk) are involved in the profile table, which belong to other units. The unit is responsible to send the signal to the

loudspeakers based on delay lines and the position of loudspeakers. This unit is not considered in this thesis and we will only discuses the filtering unit. We give a brief introduction about each of these operations from the Filter unit. In the chapter 4 we discuss this operation in more details.

1. *Bitrev* function. This function is actually responsible for generating memory indexes, which are used at the beginning of FFT operation. It is applied by *Perm* operation.
2. The other operation is *Perm* and is responsible to do a bit reversion operation since the FFT starts in bit reversion order and has to be finished in normal order.
3. The last function is *FFTld* and is responsible to perform all necessary FFT computations which are applied to the input samples. After this stage is executed, the samples are converted from time-domain into the frequency- domain or vice-versa in the case the of IFFT.

At this point we had a clear picture about which unit is a suitable candidate for a Hardware implementation. However, since our aim is to speed up the Time consuming units and the application is based on floating-point data type, we have decided to implement this operation by using a fixed point data type. The reason for this approach is because floating-point operations are expensive in terms of the hardware. They require more cycles during their computation. Besides, the computation is performed in real-time and in order to meet these real-time constraints the fixed-point data type is considered. In addition, in the DSP domain performance is sometimes more important than precision. But the speed improvement does come at the cost of reduced range and accuracy for the algorithm variables. Based on theses analysis we have decided to investigate further in order to find out how to map these requirement into the hardware. The following chapter 4 will discuses about the FFT background. In chapter 5 we have FFT unit, that utilizes a fixed-point format. Chapter 6 provides all the details of this unit regarding its organization and the utilized fixed-point format.

4 Fast Fourier Transform

4.1 Introduction

In this chapter we will first explain radix-r FFT algorithm based on Cooley-Turkey [12] decomposition. Then two architectures are explained, which can be used for implementing a FFT processor. The first is a memory-based design and the second is pipeline-based design.

4.2 Fast Fourier Transform Algorithms

The Discrete Fourier Transfer (DFT) is an important concept in many applications of digital signal processing including linear filtering, correlation analysis and spectrum analysis etc.
The DFT is defined as follows:

$$\mathrm{X[k]} = \sum_{n=0}^{N-1} X[n] W_N^{nk} \quad k=0,1,2,\ldots,\ N\ -\ 1$$

where $W_N^{nk} = e^{-j\frac{2\pi}{N}nk}$ is the FFT coefficient N = number of elements.

Figure 4.1: DFT equation

In order to directly evaluate the above equation, N complex multiplications and (N-1) complex additions for each value of the DFT are required. Since the amount of computation, and thus the computation time, is approximately proportional to N^2, this will cost a long computation time for large values of N. For this reason, it is very important to reduce the number of multiplications and additions. Fortunately, in 1965, Jim Cooley and John Tukey [12] published an efficient algorithm to compute the DFT, which is called Fast Fourier Transform (FFT) algorithm or radix-2 FFT algorithm, and it reduce the computational complexity from $\mathrm{O}(N^2)$ to $(\mathrm{N}log_N)$.

4.3 Basic Concepts of FFT Algorithms

The FFT algorithm employs two techniques to apply fast computation and reduce the number of multiplications, as described below.

4.3.1 Divide-and-Conquer

This algorithm is based on the fundamental principle of decomposing the computation of the Discrete Fourier Transfer (DFT) of a sequence of length N into successively smaller DFTs. The way in which this principle is implemented leads us to a diversity in different algorithms, all with comparable improvements in computational speed.

The divide-and-conquer can be divided in three steps namely:

1. Divide the sequence of data into two or more subsequence of smaller size.
2. Solve each subsequence recursively by the same algorithm.
3. Obtain the solution for the original problem by combining the solutions to the subsequences.

4.3.2 Properties of the Coefficient(W_N^{nk})

The main control parameter in equation (4.1) is the W_N^{nk}. it is the DFT coefficient also called twiddle factor. The partial twiddle factor of an N- point DFT (N is the power of two) is shown in Figure 4.2.

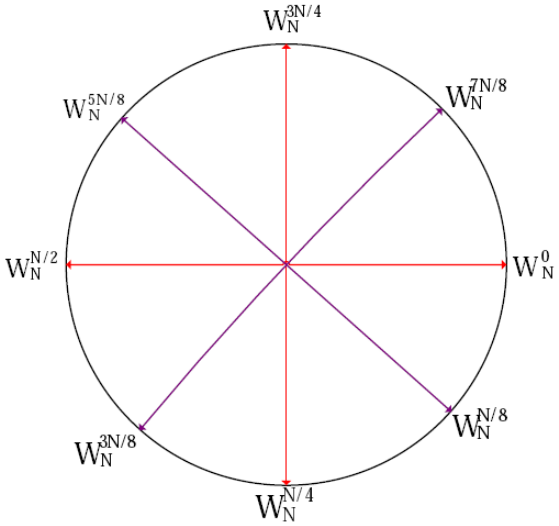

Figure 4.2: The partial twiddle factor of an N-point DFT.

To improve the efficiency of the computation of the DFT, most approaches apply the properties of twiddle factor (W_N^{nk}) as bellow:

1. $W_N^0 = -W_N^{N/2} = 1$

   $W_N^{N/4} = -W_N^{3N/4} = -j$
2. $W_N^{n+(k+N/2)} = -W_N^{nk}$ *Symmetry*
3. $W_N^{nk} = -W_N^{(n+N)k} = W_N^{n(N+k)}$ *Periodicity in n and k*

The fact that certain values of the product k.n, the sin and cosine functions take on the value 1 or 0 (property 1) lead us to eliminating the multiplications, as shown below:

$$A * W_N^{nk} + B * W_N^{nk+N/2} = (A + B * W_N^{N/2}) * W_N^{nk} = (A - B) * W_N^{nk} \quad (4.1)$$

table $A * W_N^{nk} + B * W_N^{nk+N/4} = (A + B * W_N^{N/4}) * W_N^{nk} = (A - jB) * W_N^{nk}$ (4.2)

Unfortunately, this approach still leaves us with an amount of computation that is proportional to N^2. However to achieve a greater reduction of the computation, the second property of the periodicity of the complex sequence is applied as is shown below:

A * W_N^{nk} + B * $W_N^{n(k+N)}$ = (A + B * W_N^{nN}) * W_N^{nk} = (A + B) * W_N^{nk} (4.3)

where W_N^{nN} = 1 n= 0, . . . , N - 1.

The symmetric property of the twiddle factor can further be exploited as follow:

$W_N^{nk+N/8}$ = -$W_N^{nk+5N/8}$ = $\frac{\sqrt{2}}{2}$(1-j)*W_N^{nk} and $W_N^{nk+3N/8}$ = -$\frac{\sqrt{2}}{2}$(1+j)*W_N^{nk} plugging these

equations to 4.3 we have:

A * W_N^{nk} + B * $W_N^{nk+N/8}$ = (A+B * $W_N^{N/8}$) * W_N^{nk} = {A + $\frac{\sqrt{2}}{2}$(1 - j)B} * W_N^{nk}

$\frac{\sqrt{2}}{2}$(1-j)B = $\frac{\sqrt{2}}{2}$(1-j)*(a + jb)=$\frac{\sqrt{2}}{2}$\{(a+b) + j(b-a)\} (4.4)

A*W_N^{nk} + B * $W_N^{nk+3N/8}$ = (A + B * $W_N^{3n/8}$) * W_N^{nk} = {A - $\frac{\sqrt{2}}{2}$(1+j)B} * W_N^{nk}

-$\frac{\sqrt{2}}{2}$(1+j)B=-$\frac{\sqrt{2}}{2}$(1+j)*(a+jb) = $\frac{\sqrt{2}}{2}$\{(b - a) - j(a+B)\} (4.5)

The multiplication of a complex number from (4.4) and (4.5) costs two Real multiplication and two Real additions. Thus, we have the symmetry utilization of phase difference ± 45° only costs two Real multiplications.

4.4 Classification of FFT Algorithms

FFT algorithm can be divided into three classes namely: fixed-radix, split-radix and mixed-radix. However during this section we will only cover the fixed-radix class.

The FFT algorithm consists of two basic approaches. The first one is called decimation in-time (DIT). The second one is called decimation-in-frequency (DIF). So far, all mentioned approaches from the previous chapters were based on the former type. Our FFT design is based on decimation-in-frequency type. However, both of these types are based on the recursive decomposition of an N-point transform into successively smaller subsequences. There is no difference in computational complexity between these two types of algorithms. The only difference between these two algorithms is that, DIT starts with bit reversed order input and generates normal order output. Nevertheless, DIF starts with normal order input and generates bit reversed order output. Only the decimation-in-frequency (DIF) will be considered here.

4.4.1 Fixed-Radix FFT Algorithms

In this section we will introduce different fixed-radix FFT algorithms such as radix-2, radix-4 and radix2^2.

4.4.2 Radix-2 FFT Algorithm

The radix-2 FFT algorithm is obtained by using the divide-and-conquer approach. To derive the algorithm, we begin by dividing the DFT formula into two summations, one of which contains the sum over the first N/2 data points and the second sum contains the last N/2 data points. Thus we obtain:

$$\begin{aligned} X(k) &= \sum_{n=0}^{(N/2)-1} x(n) W_N^{nk} + \sum_{n=N/2}^{N-1} x(n) W_N^{nk} \\ &= \sum_{n=0}^{(N/2)-1} x(n) W_N^{nk} + \sum_{n=0}^{(N/2)-1} x(n+\frac{N}{2}) W_N^{(n+N/2)k} \\ &= \sum_{n=0}^{(N/2)-1} x(n) W_N^{nk} + W_N^{kN/2} \sum_{n=0}^{(N/2)-1} x(n+\frac{N}{2}) W_N^{nk} \end{aligned}$$

Since $W_N^{kN/2}$ = -1, the equation can be rewritten as:

X(k) = $\sum_{n=0}^{(N/2-1)} [x[n] + (-1^k) x(n + \frac{N}{2}] W_N^{nk}$ (4.6)

Now, let us again restrict the discussion to N a power of 2 and consider dividing of computation into the even-numbered frequency samples and the odd-numbered frequency samples. Thus we obtain the even-numbered frequency samples as

X[2k] = $\sum_{n=0}^{(N/2)-1} [x(n) + (-1)^{2k} x(n + \frac{N}{2})] W_N^{n2k}$ k = 0,1,..., $\frac{N}{2}$ - 1 (4.7)

= $\sum_{n=0}^{(N/2)-1} [x(n) + x(n + \frac{N}{2})] W_{N/2}^{nk}$

The above equation94.7) is the N/2 point DFT of the N/2 point sequence obtained by adding the upper half and the bottom half of the input sequence.

The same approach can be applied to consider the odd-numbered frequency samples as below

X[2k +1] = $\sum_{n=0}^{(N/2)-1}[x(n) + (-1)^{2k+1}x(n + \frac{N}{2})]W_N^{n2k+1}$ k = 0,1,...,$\frac{N}{2}$ - 1 (4.8)

Where we have used the fact that $W_N^{n2k+1} = W_N^n * W_{N/2}^{nk} and (-1)^{2k+1} = -1$

Substituting into Equation (4.8), we obtain

X[2k +1] = $\sum_{n=0}^{(N/2)-1}[x(n) - x(n + \frac{N}{2})]W_N^n W_{N/2}^{nk}$ k = 1,..., $\frac{N}{2} - 1$ (4.9)

The above equation (4.9) is the N/2 point DFT of the N/2 point sequence. This is obtained by first subtracting the bottom half of the input sequence from the upper half. After that, the resulting sequence is multiplied by W_N^n. if we define the N/2 point sequences g (n) and h(n) as

g(n) = x(n) + x(n + $\frac{N}{2}$) n = 0,1,...,$\frac{N}{2}$ - 1

h(n) = x(n) - x(n + $\frac{N}{2}$) (4.10)

then

X(2k) = $\sum_{n=0}^{(N/2)-1} g(n)W_{N/2}^{nk}$

X(2k + 1) = $\sum_{n=0}^{(N/2)-1} g(n)W_{N/2}^{nk}W_N^n$ (4.11)

The computation of the sequence g(n) and h(n) according to Equation (4.11) and the subsequent use of these sequences to compute the N/2 point are shown in figure 4.3. For the 16-point DFT, the computation has been reduced to a computation of 2-point DFTs. The 2-point DFT of, for example, x(0) and x(8) are depicted in figure 4.4.

If we substitute the $\frac{N}{2}$-point DFT boxes of figure 4.4 with the figure 4.5, then we can see the complete 16-point DFT graph in figure 6.6.

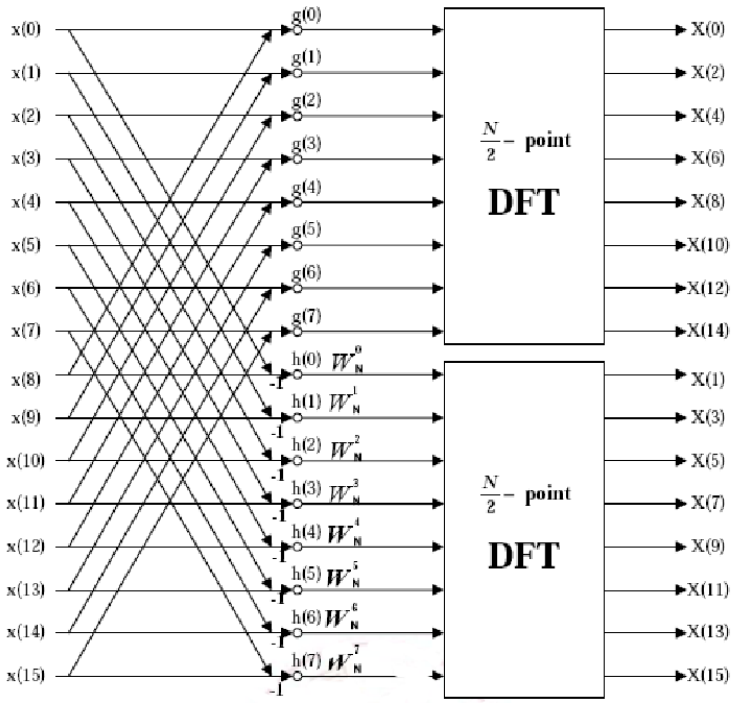

Figure 4.3: First stage of the radix-2 DIF FFT algorithm

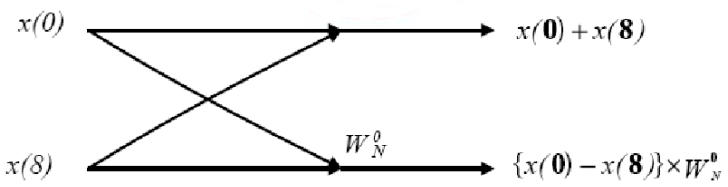

Figure 4.4: Signal flow graph of a typical 2-point DFT

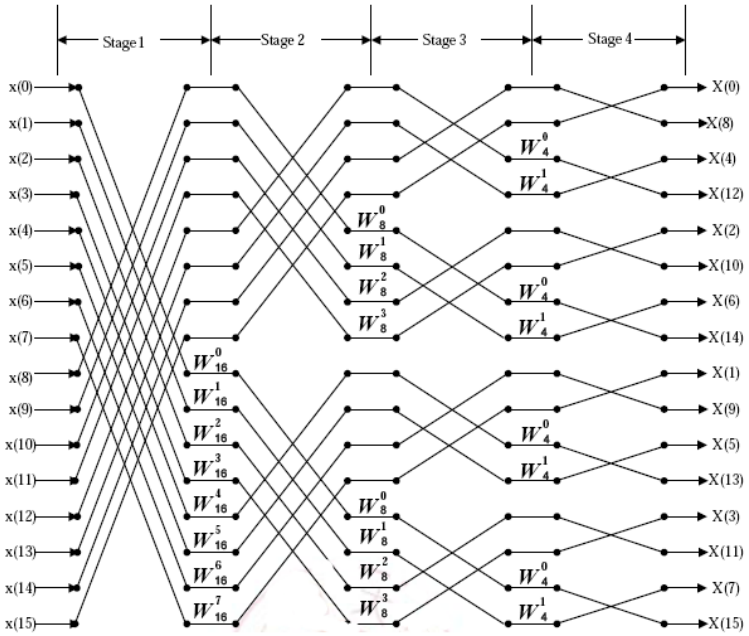

Figure 4.5: Radix-2 DIF FFT signal flow graph of 16-point

Figure 4.6 shows the signal flow from one stage to the next. The basic computation, which is depicted in figure 4.5 involves obtaining a pair of values in one stage from a pair of values in the preceding stage. During this stage the coefficients are always power of W_n and the exponents are separated by N/2. The signal flow graph in figure 4.4 is also referenced as a butterfly. We also notice that the butterfly number of N/2 is regular in each stage. In addition, by modifying the figure 4.5 as it is shown in figure 4.7, this lead to only one complex multiplication and two complex additions.

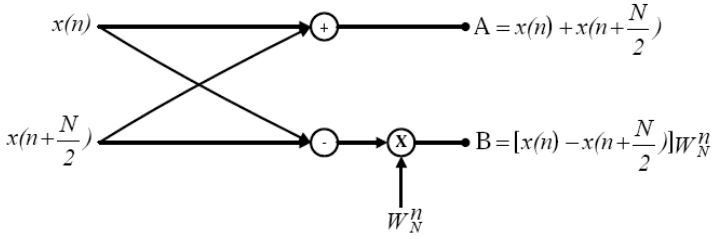

Figure 4.6: The basic butterfly for radix-2 DIF FFT algorithm.

By counting the arithmetic operations in figure 4.6 and generalizing to N = $2^v (v = log_2 N)$ stages of decimation, which each stage involves N/2 butterflies of the type shown in figure 4.7. Thus, the computation of the N-point DFT via the decimation-in-frequency FFT algorithm, requires (N/2)$log_2 N$ complex multiplications and Nlog_nN complex additions.

We observe from figure 4.6, that the time domain input data x(n) occurs in natural order, but the frequency domain output DFT X(k) occurs in bit-reversed order. We also note that the computations are performed in-place. In-place processing means that memory reads and memory writes in each butterfly processing use the same memory location. In this way the required memory space can be minimized. We also observe from figure 4.6 the relationship between input and output data is that index $[klog_{N-1}klog_{N-2}...k_2k_1]2$ is mapped to index $[k0k1, ..., klog_2N - 2klog_2N - 1]2$ in a one dimension memory array. For instance, in the 16-point radix-2 DIF FFT signal flow graph, the output index 11_{10} or 1011_2 is mapped to index 13_{10} or 1101_2 of the memory array. For the radix-2 16-point DIF FFT signal flow graph, the relationship between normal order and bit-reversed order is depicted in figure 4.8.

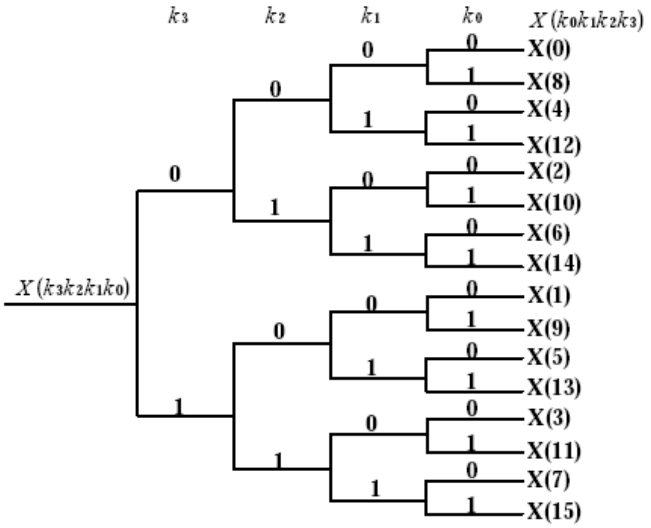

Figure 4.7: Four diagram depicting bit-reversed sorting [13].

In addition, it is possible to reorder the decimation-in-frequency algorithm so that the input sequence occurs in bit-reversed order while the output DFT occurs in normal order. Furthermore, if we abandon the requirement that the computations should done in place, it is also possible to have both the input data and the output DFT in normal order.This case is called out-of-place mode.

4.4.3 Radix-4 FFT Algorithm

When the number of data points N in the DFT is a power of 4 (i.e., $N = 4^v$)), we can always use a radix-2 algorithm for the computation. However, for this case, it is more efficient computationally to employ a radix-4 FFT algorithm.

Similarly to the radix-2 FFT equation we use divide-and-conquer technique to accomplish the N-point DFT into four N/4 point DFTs. We have

$$\begin{aligned} X(k) &= \sum_{n=0}^{N-1} x(n) W_N^{kn} \\ &= \sum_{n=0}^{N/4-1} x(n) W_N^{kn} + \sum_{n=N/4}^{N/2-1} x(n) W_N^{kn} + \sum_{n=N/2}^{3N/4-1} x(n) W_N^{kn} + \sum_{n=3N/4}^{N-1} x(n) W_N^{kn} \\ &= \sum_{n=0}^{N/4-1} x(n) W_N^{kn} + W_N^{Nk/4} \sum_{n=0}^{N/4-1} x(n+\frac{N}{4}) W_N^{kn} \\ &\quad + W_N^{Nk/2} \sum_{n=0}^{N/4-1} x(n+\frac{N}{2}) W_N^{kn} + W_N^{3Nk/4} \sum_{n=0}^{N/4-1} x(n+\frac{3N}{4}) W_N^{kn} \end{aligned}$$

From the definition of the twiddle factors, we have

$W_N^{Nk/4}=(-j)^k$, $W_N^{Nk/2}=(-1)^k$, $W_N^{3Nk/4}=(j)^k$ (4.12)

After substitution (5.12) into (5.11), we obtain

$X(k) = W_{n=0}^{Nk/4-1}$ [x(n) + $(-j)^k x(n+(\frac{N}{4}) + (-1)^k x(n+\frac{N}{2}) + (j)^k x(n+\frac{3N}{4})]W_N^{nk}$ (4.13)

The relation is not an N/4-point DFT because the twiddle factor depends on N and not on N/4. To convert it into an N/4-point DFT we subdivide the DFT sequence into four N/4-point subsequences, X(4k), X(4k+1), X(4k+2), and X(4k+3), k = 0, 1, ..., N/4. Thus we obtain the radix-4 decimation-in frequency DFT as

$$\begin{aligned} X(4k) &= \sum_{n=0}^{N/4-1} \left[x(n) + x\left(n+\frac{N}{4}\right) + x\left(n+\frac{N}{2}\right) + x\left(n+\frac{3N}{4}\right) \right] W_N^0 W_{N/4}^{kn} \\ X(4k+1) &= \sum_{n=0}^{N/4-1} \left[x(n) - jx\left(n+\frac{N}{4}\right) - x\left(n+\frac{N}{2}\right) + jx\left(n+\frac{3N}{4}\right) \right] W_N^n W_{N/4}^{kn} \\ X(4k+2) &= \sum_{n=0}^{N/4-1} \left[x(n) - x\left(n+\frac{N}{4}\right) + x\left(n+\frac{N}{2}\right) - x\left(n+\frac{3N}{4}\right) \right] W_N^{2n} W_{N/4}^{kn} \\ X(4k+3) &= \sum_{n=0}^{N/4-1} \left[x(n) + jx\left(n+\frac{N}{4}\right) - x\left(n+\frac{N}{2}\right) - jx\left(n+\frac{3N}{4}\right) \right] W_N^{3n} W_{N/4}^{kn} \end{aligned}$$

where we have used the property $W_N^{4kn} = W_{N/4}^{kn}$. Note that the input to each N/4-point DFT is a linear combination of four signal samples scaled by a twiddle factor. This procedure is repeated v times, where v = log_4N. The complete butterfly operation for Radix-4 DIF is shown in figure 4.11 (a) and in a more compact form in figure 4.11 (b).

The cost of the computation for the algorithm is 3vN/4 = (3N/8) $log_2 N$ complex multiplications and (3N/42)$log_2 N$ complex additions. We note that the number of multiplications

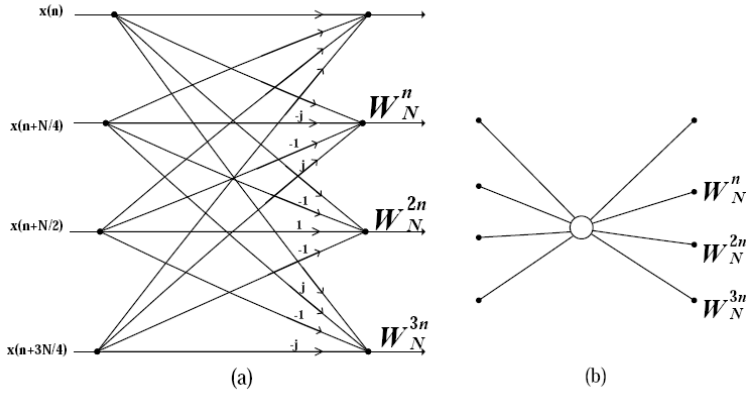

Figure 4.8: The basic butterfly for radix-4 DIF FFT algorithm.

is reduced by 25%, however but the number of additions has increased 50% from N$log_2 N$ to (3N/2)$log_2 N$.

Figure 4.12 depicts a 16-point radix-4 DIF. We show only one of the 4-point DFTs in the first stage, otherwise, the graph will become too complicated.

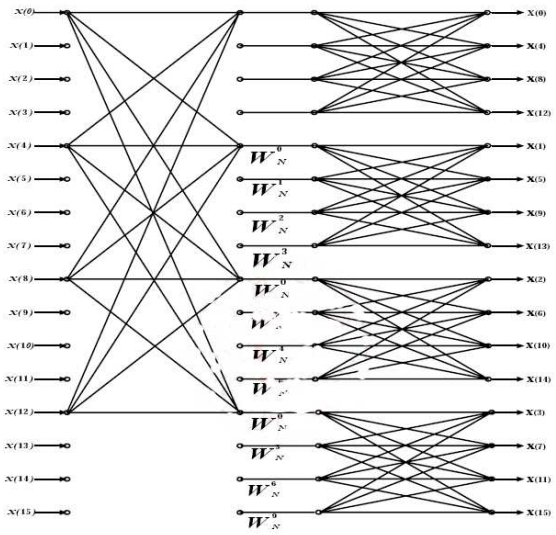

Figure 4.9: Radix-4 signal flow graph.

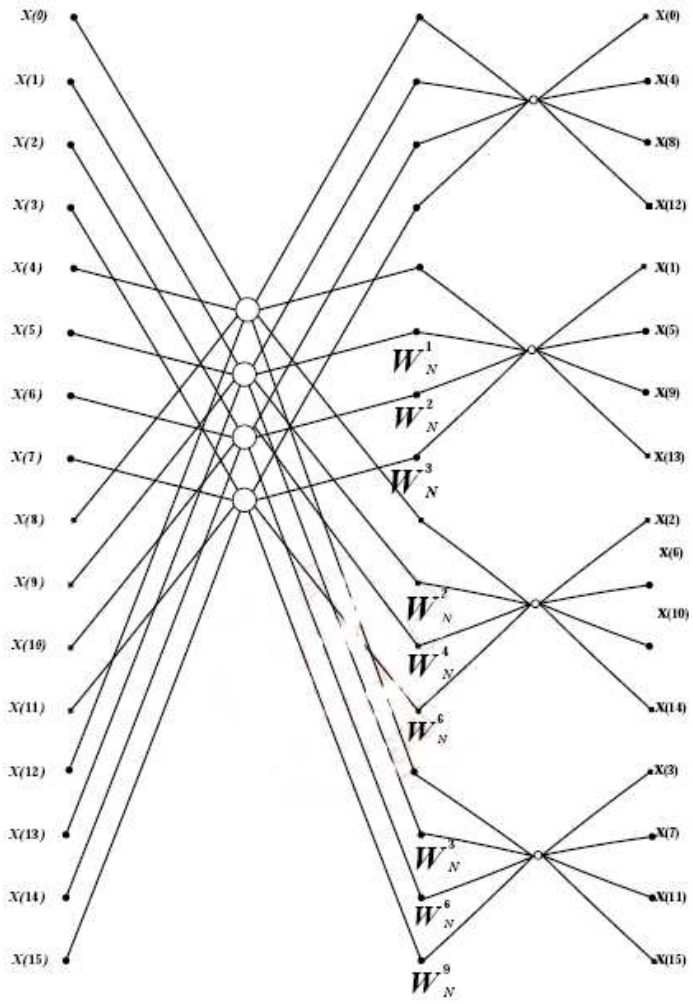

Figure 4.10: Radix-4 signal flow graph.

Figure 4.13 shows the radix-4 DIF FFT algorithm, where its input is in normal order and its output is in digit-reversed order. The in-place computation is used in radix-4 FFT algorithm. The signal flow graph of the radix-4 16-point DIF FFT is shown in figure 4.14, Where it's also depicted the relationship between normal order and digit-reversed order with two digits of k_0 and $k_1 \in \{0, 1, 2, 3\}$.

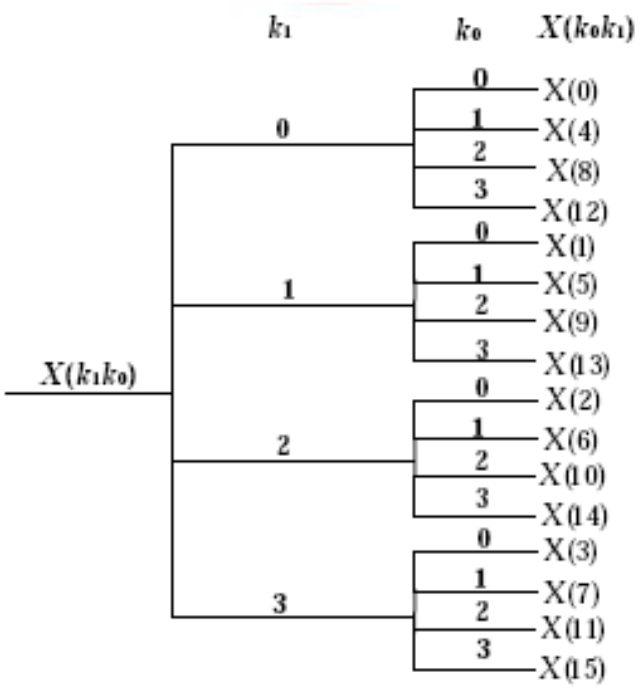

Figure 4.11: Four diagram depicting digit-reversed sorting.

4.4.4 Radix-2^2 DIF FFT Algorithm

In case where N = 2^{2l}, instead of using radix-2 algorithms, we can use radix-4 FFT algorithms. This leads to having additional savings in the required number of complex multiplications. The derivation of the radix-4 algorithms is similar to radix-2 ones As it was explained in the previous section, the radix-4 FFT algorithm needs 3N complex additions. However, the number of additions in the radix-4 basic cell can be decreased. Figure 4.15 shows an improved basic cell for the radix-4 algorithm that required fewer operations. The number of complex additions is decreased to 2N per stage, instead of 3N. This approach is called $radix - 2^2$ fft algorithm.

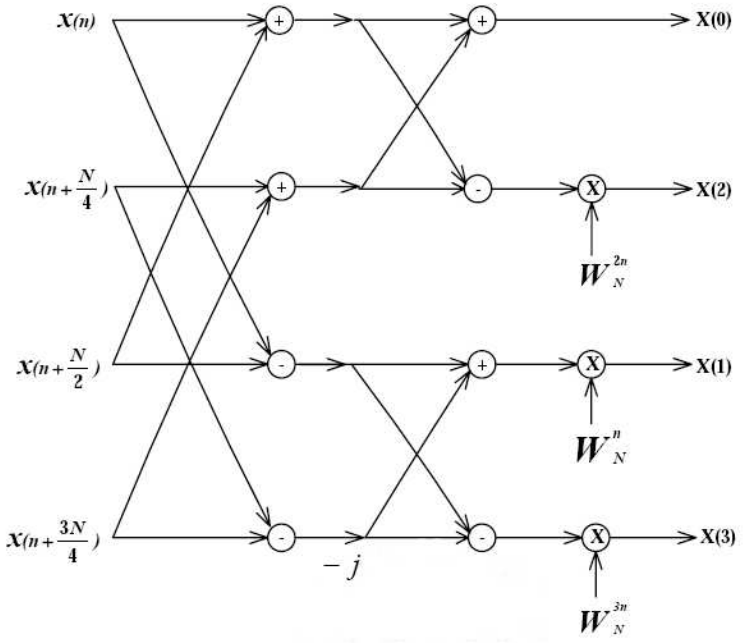

Figure 4.12: The basic butterfly for radix-22 DIF FFT algorithm.

In the figure 4.15 we have:

$X(0) = x(n) + x(n + \frac{N}{4}) + x(n + \frac{N}{2}) + x(n + \frac{3N}{4})$

$X(2) = \{x(n) - x(n + \frac{N}{4}) + x(n + \frac{N}{2}) - x(n + \frac{3N}{4})\}W_N^{2n}$

$X(1) = \{x(n) - x(n + \frac{N}{4}) - jx(n + \frac{N}{2}) + jx(n + \frac{3N}{4})\}W_N^{n}$

$X(3) = \{x(n) = jx(n + \frac{N}{4}) - x(n + \frac{N}{2}) - jx(n + \frac{3N}{4})\}W_N^{3n}$

The complete signal flow graph of 16-point radix-2^2 DIF FFT is depicted in Figure 4.16.

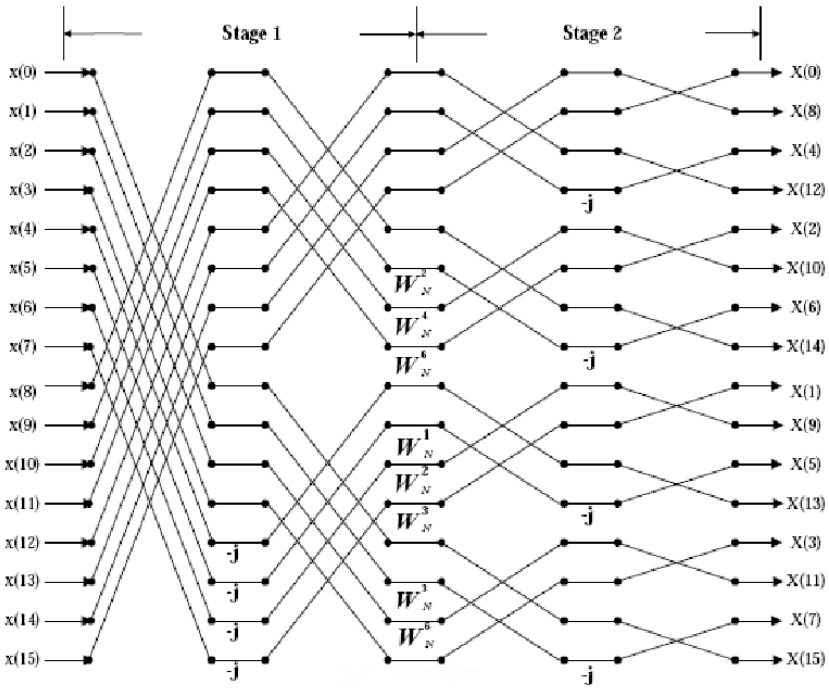

Figure 4.13: Four diagram depicting digit-reversed sorting.

As shown in figure 4.15, there are N/4 butterflies for each stage with good regularity. During this stage we also apply bit-reversed ordering to the output data as in the radix-2 DIF FFT algorithm. In addition, the radix-2^2 FFT algorithm can be considered as a modification of the radix-2 FFT algorithm, which reorders the twiddle factor of two radix-2 butterfly stages.

4.4.5 Summery

In this section we have described two FFT algorithms. We have seen that each one of them has different property regarding computational complexity and butterfly architecture. According to the references [13-14-15], we have shown in table 4.1, a comparison between the two FFT algorithms in terms of complex additions and multiplications. Table 4.1 depicts the number of multiplications and additions for the complex numbers. However, for the Real numbers different schemes are applied:

1. Two Real additions and four Real multiplications
2. Three Real additions and three Real multiplications

   One complex addition is done using two Real additions. Only two Real multiplications and two Real additions cost, if the twiddle factor has phase difference ± 45°.

FFT Algorithm	Number of complex multiplications	Number of complex additions
Radix-2	$\frac{N}{2}log_2N - N + 1$	N log_2 N
Radix-4	$\frac{3N}{2}log_2N - N + 1$	N log_2 N

Table 4.1: comparisons of FFT algorithms

4.5 FFT HARDWARE ARCHITECTURES

In the previous section we have described two different FFT algorithms. These all are extended from Cooley-Tukeys radix-2 FFT algorithm [12] which reduce the complexity of the algorithm from O(N^2) to O(N log N) During this section, our aim is to describe the hardware architecture of an FFT system.

The hardware architecture can be divided into two classes namely: pipeline-based architecture [16]-[17] and memory-based design [18]-[19]. In this thesis we have implemented the DIF FFT processor with memory-based architecture.

4.5.1 Pipeline-Based FFT Architecture

Pipeline processor belongs to one of the two architectures, that can be applied for application specific real-time DFT computation utilizing fast algorithms. This architecture is highly regular and can be easily scaled and parameterized when using Hardware Description Language (HDL).

In addition, this type of FFT architectures offers more flexibility when transforms of different lengths are to be computed with the same chip. Figure 4.17 depicts a 16-point radix DIF FFT example of projection mapping. It consists of several butterfly processing elements (BPE) where each one of them includes complex multipliers and some data buffers. Two types of data buffering strategies [20,17,21,22] for the pipeline-based FFT processor. These strategies can be divided as follows:

- Delay-commutator (DC) architecture
- Delay-feedback (DF) architecture

A Multi-path structure is used with delay-commutator type whereas single-path structure is usually used with delay-feedback type. The former strategy is also called multi-path delay-commutator (MDC) and later single-path delay-feedback (SDF). In the next subsections we will discuses these in more details.

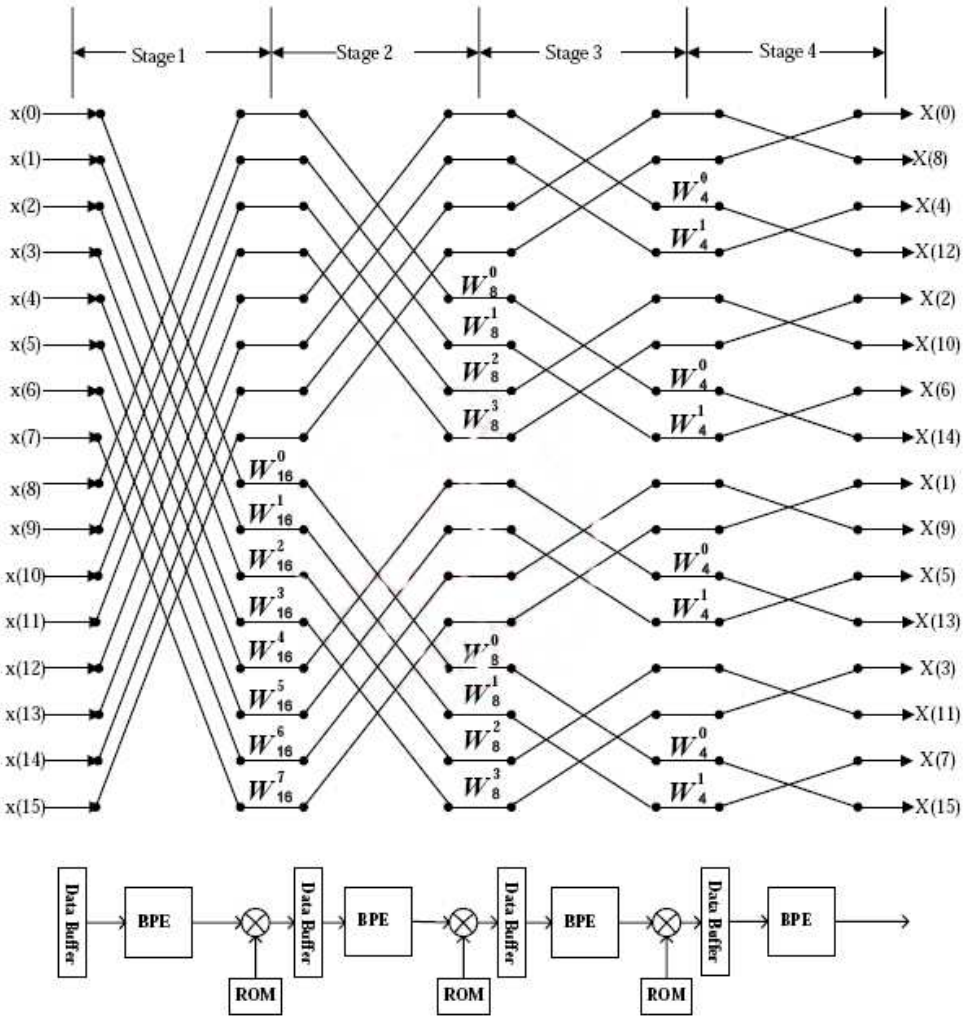

Figure 4.14: Projection mapping of the radix-2 DIF signal flow graph.

4.5.1.1 Single-Path Delay-Feedback Pipeline Architecture

The single-path delay-feedback (SDF) pipeline architecture can be applied for several FFT algorithms such as radix-2 and radix-4. However we will only introduce radix-2, since the others are all similar to radix-2. For instance, figure 4.18 shows a 16-point radix-2 DIF SDF pipelined FFT processor architecture. The SDF architecture employs a feedback register as a storage area for the butterfly during the processing stage. Only a part of the data can be processed during the processing stage. The utilization rate of adders and multipliers for radix-2 algorithm is equal to 50%. The hardware utilization of SDF for radix-2 based on a N-point SDF FFT architecture can be categorized as follows:

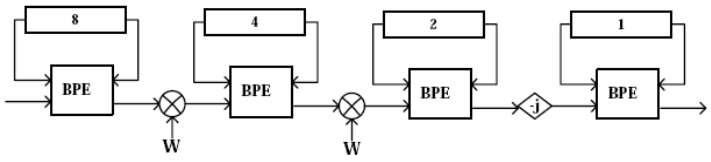

Figure 4.15: Radix-2 Single-path Delay Feedback (N=16).

- Number of required Adders: $2log_nN$
- Number of required Multipliers: $log_2N - 2$
- Number of required shift registers:ShiftRegisters N - 1

As an example figure 4.18 shows a 16-point SDF FFT architecture that uses three complex multipliers, four butterfly processing elements and fifteen shift registers.

4.5.1.2 Multi-Path Delay Commutator Pipeline Architecture

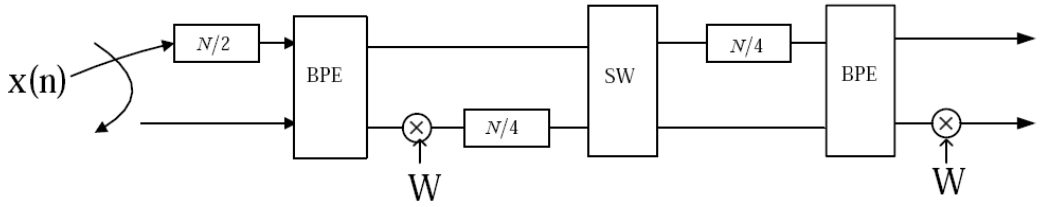

Figure 4.16: The first two stages of N-point radix-2 MDC structure.

In case of Multi-Path Delay(MDC) intermediate data are sent to output of the next stage or coefficient multiplier instead of being written back as it was the case in SDF architecture. During this stage the data has to be divided into two parallel data streams. Each one will enters the butterfly processing element according to proper delay time of MDC algorithm. In this case we also use radix-2 DIF FFT algorithm to introduce MDC-based FFT processor architecture.

Figure 4.19 is depicts the first two stages of N-point radix-2 MDC-based FFT processor. From figure 4.19 we can observe that there are numbers of elements involved between the PEs(processing elements). These components can be divided into a set of shift registers and a commutating switch. The aim of these components are to establish a suitable set of data for the next PE. From an N-point MDC FFT architecture the number of required hardware components are estimated as follows:

- Number of required Adders: $2log_2N$
- Number of required Multipliers: $log_2N - 2$
- Number of required shift registers: $\frac{3N}{2} - 2$

The utilization of these components is equal to 50%. For instance when, N=16 then 2 complex multipliers, 4 butterfly processing elements and 22 shift registers are required.

4.5.1.3 Comparison of the Pipeline-Based FFT processor

Table 4.2 gives an overview of the required components for two architectures. It also mention the hardware utilization of the multipliers, adders, butterfly processing elements (PE) and memory (register). Finally, the results of these two architectures are compared and are proved that the SDF structure needs less hardware requirement than the other structure.

Algorithm		Radix-2	
Architecture		SDF	MDC
Hardware utilization	Multiplier	50%	50%
	PEs	50%	50%
Hardware requirement	Multiplier	$log_2(N) - 2$	$log_2(N) - 2$
	PE	$log_2(N)$	$log_2(N)$
	Memory	N - 1	$\frac{3N}{2} - 2$

Table 4.2: The hardware utilization and requirement in different structure

4.5.2 Memory-Based FFT Architecture

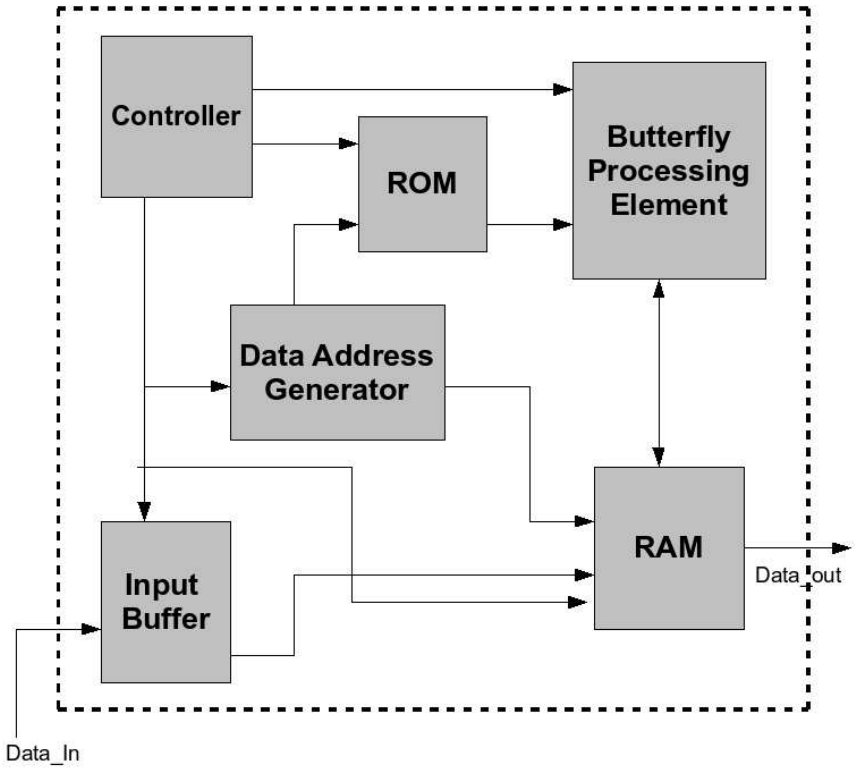

Figure 4.17: The first two stages of N-point radix-2 MDC structure.

The aim of Memory-based FFT processor architectures are to increase the utilization rate of butterfly processing elements (PE) and reduce the redundancy of multiple butterflies PEs. They usually contain one or two large memory blocks. All data should be accessed from a centralized memory block, in contrast to pipeline architectures, which utilize small distributed memory blocks that cooperate with local arithmetic units. Usually as illustrated in figure 4.20, the architecture of a memory-based can be divided into the number of subunits:

- Controller
- Coefficient ROM
- Processing Element (Butterfly)
- Main Memory (RAM)
- A set of buffers for buffering the input data

Memory based architectures can be divided into several types, however the main difference, which distinguish between them comes from their butterfly processing element and the buffering strategies. These strategies are employed during the intermediate results produced in the FFT computation. Numerous algorithms can be applied for this architecture. These algorithms are radix-2, radix-4, radix-8 or other radix butterfly PEs. Several buffering strategies exist and can be applied during this stage. Buffering strategies can be divided as follows:

- In-place mode [18]-[15]
- Out-of-place mode (ping-pong mode [26])
- Other modes

In general, if the data of butterfly PE during processing are written back to the same memory addresses of input data, this design is called in-place architecture. However, when output data are written to another memory block without overwriting input data, it is called an out-of-place design.

The controller of a memory-based FFT processor is responsible to activate and deactivate all necessary control signals during each computation phase. Another important unit is the data address generator which generates the correct indices for ROM coefficients and also for the main memory. A conventional processing order and control scheme for radix-2 FFT was first introduced by Cohen [27]. This concept was then extended and generalized by Ma [18][28] and Lo [23]. The main memory contains all data, which are used by the butterfly. The butterfly starts by reading the data from the main memory and perform the necessary computations to transform them to the frequency domain. Finally, the results are stored back into the main memory according to the addresses from data address generator. During the computations only one memory bank can be accessed due to the in-place mode being adopted for buffering strategy. However to speed up the memory access bandwidth, two alternatives can be considered:

- Use a multi-port Memory.
- Partition a memory bank into separate banks, thus access more elements at the same time.

The coefficient index generator is responsible to generate indices of coefficients for the ROM memory. This unit uses the stage number and butterfly number to calculate the required coefficient data.

Operation state controller is a counter-based controller and is responsible to monitor the state of the processor and perform a desired operation. A butterfly processing element (PE) consists of complex adders and complex multipliers. However, the design architecture of PE depends on what type of FFT algorithm is used.

4.5.3 Summary

In this section two different classes were described for designing the hardware architecture of FFT processor. Pipeline-based architecture consists of MDC and SDF. These architectures offer a continuous data throughput with simple control strategy. However, they require more hardware resources. In contrast to the pipeline architectures, memory-based architectures require less hardware arithmetic units. In addition, Memory-based architectures have an advantage of low memory cost and low PE cost. In contrast to pipeline-based architecture, the memory-based architecture implements one centralized memory instead of distributed small-size memory blocks or shift registers. This leads to having much more hardware resources for pipeline architecture. Apart from that, memory-based architecture is also offering a low number of butterflies PEs. This architecture is very suitable for long length FFT implementations, however the down side of this concept is a lower throughput. In general, pipeline-based architecture offers a high throughput with a high cost of hardware resources, whereas memory-based architecture offer a reasonable performance with a low area utilization.

5 Fixed-Point Representation

5.1 Introduction

More accurately to construct an algorithm double or signal precision floating-point data must be used. However, there is an overhead, which is caused during the execution of floating point calculations. This phenomenon limits the effective iteration rate of an algorithm.
To improve throughput or increase the execution rate, calculations can be performed using two's complement signed fixed point representation [3]. Fixed-point representation is established by creating a virtual decimal place in between two bit locations for a given length of data.
For the purpose of this thesis the notation of a Q-point for a fixed-point number is introduced as follows:

$$Q[QI]./[Qf]$$

Where QI=#of integer number bits and QF=#of fraction bits

The number of integer bits (QI) plus the number of fractional bits (QF) is equal to the total number of bits used to represent the number. Sum (QI+QF) is known as Word Length and this sum corresponds to word width supported on a given processor. Typical word lengths would be 8, 16, 32 bits corresponding to char, short int, int.

5.2 Fixed-Point Range - Integer Portion

To represent a floating-point number in fixed-point a floating-point number needs to be viewed as two distinct parts, integer content and the fractional content. The integer range of a floating-point variable in an algorithm sets the number of bits (QI) required to represent the integer portion of the number. The following method is used to computing the number of integer bits required (QI) for each type of number.

$$(-2^{QI-1}) \leq \alpha \leq (2^{QI-1})$$

(5.0)

The above formula (5.0) can be used for negative as well as positive value's. Formula (5.2) shows how it can be used for a negative constraint.

$$\begin{aligned}(-2^{QI-1}) &\leq \alpha \\ -2^{QI-1} &\geq -\alpha \\ QI - 1 &\geq log_2(-\alpha) \\ QI &\geq log_2(-\alpha) + 1\end{aligned}$$

(5.1)

The following formula (5.2) is used for a positive constraint:

$$\begin{aligned}\alpha &\leq (-2^{QI-1}) \\ \alpha + 1 &\leq 2^{QI-1} \\ log_2(\alpha + 1) &\leq QI - 1 \\ QI &\geq (log_2(\alpha + 1)) + 1\end{aligned}$$

(5.2)

5.3 Fixed-Point Resolution - Fractional Portion

The resolution for a fixed-point variable is set by the number of fractional bits (QF) used in fixed-point variable. The resolution ϵ, of a fixed-point number is shown in formula 5.3:

$\epsilon = \frac{1}{2^{QF}}$ (5.3)

Therefore, the number of fractional bits (QF) required for a particular resolution are defined by the equation:

$$\text{QF} = log_2(\frac{1}{\epsilon})$$

However, since QF is integer values only, the ceiling of logarithm is used:

$$\text{QF} = \lceil (log_2(\frac{1}{\epsilon})) \rceil$$

In the next section we will discuss how these equation are applied to determine number of bits, which are used to encode the FFT unit.

5.4 Converting FFT unit From Floating-point into Fixed-point

In this section we will explain about the process of converting a Floating Point FFT Unit into a Fixed-point. As it was discussed in the previous chapter the reason for this process is to speed up the execution, since fixed-point arithmetic operations are faster then the floating-point ones. This process will increase the data throughput, since the FFT unit requires fewer cycles during each operation. However this process causes a negative effect on data representation, since fixed-point numbers are less accurate compared with Floating point data. Figure 5.4 shows how the FFT unit is buildup. In the previous chapter we also mentioned these methods, however in this section we will give more details about each of these methods. Furthermore, we will emphasize on the data representation and specially discuss the conversion from floating point to fixed-point by using the equations mentioned in the previous section.

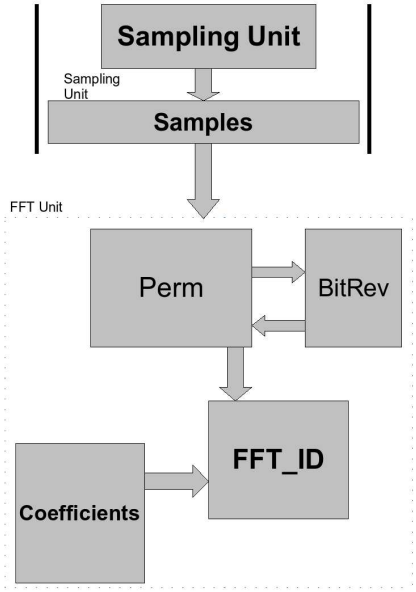

Figure 5.1: FFT Unit Overview

As it is shown in figure 5.1, the FFT unit consists of four sub units. Each of these subunits is responsible for doing a certain task. In this case each subunit will execute its task sequentially, which be considered as the unit bottleneck. This is actually an important issue and will be discussed in more detail in the upcoming chapters.

The sampling unit is very important for the FFT and data conversion. This unit is responsible for taking samples from a continuous signal. Most of the signals directly encountered in science and engineering are continuous: light intensity that changes with distance; voltage that varies over time, etc. Analog-to-Digital Conversion (ADC) is the process that allows digital computers to interact with these everyday signals. Digital information is different from its continuous counterpart in two important respects: it is sampled, and it is quantized. Both of these restrict how much information a digital signal can contain. The job of this unit is to collect samples from a dry signal and writing them into the memory.

The sampling frequency is setup to 44100 (44.1 KHz). This is equal to taking a sample every 23 μsec. The WFS application uses a Wav file as an input to sampling unit. The samples are taken from the file and stored into the memory. The number of input channels for this application is equal to one and the number of audio samples is equal to 22050. After the memory is loaded with these samples, these samples need to be processed by other units for further processing. However, this unit can also be considered as another bottleneck of this application, since only one sampling unit involved in this process. However by using more channels the data throughput can significantly be improved [4]. This approach is not treated further in this thesis and we will only concentrate on data conversion in this chapter.

During this process the samples are captured in a floating point data type, however each sample contains only information in their integer part. This means that useful information is only resided in integer part and the fractional part is equal to zero. When integer part was analyzed we discover that this part is perfectly representable by using only 16-bits (Short int) of data. Thus, at this stage we can easily modify the code in such a way that the integer part can be captured in 16-bits (short int) without considering the fractional part.
Since the fractional part contains only zeros, we decide to remove this part, which leads us to capturing samples in only 16-bits instead of a floating-point data type. We knew that the sampling process specially performed for audio processing usually uses 16-bits of data, although there are some systems, which use 24-bits instead of 16-bits. However in our case only 16-bits were enough to capture each sample. At this point the fixed point conversion was performed for sampling unit and each sample is presented by 16 bits for further processing.

After this stage, our next aim was to investigate how to apply the same process for the other part of the FFT unit in order to perform the whole calculation in fixed-point arithmetic. As we have seen from the previous figure 5.1, FFT unit consists of the following subunits namely:

- Perm unit
- BitRev Unit
- FFTld Unit
- Coefficient Unit

Perm is the first subunit in FFT, which is used to store the samples in memory according to computed indices during this stage. Indices are generated by BitRev unit. The aim of this unit is the reversing of the bits in a binary word from left to right. Therefore the MSB become LSB and the vice versa. However, in the case of data conversion these two units(Perm and BitRev) were modified, since the aim of these unit is to generate indices for these samples, so this information can be kept into the memory according to their indices.

The next unit is FFTld and this unit is actually the core of FFT transform. This unit is responsible to convert the signals from Time-domain into frequency-domain. It does that by first reading the samples from the memory and multiplies these with coefficients. As it was already explained samples are represented in 16-bits. However, coefficients are represented in floating-point and need to be represented in fixed-point as well in order to do the computation in this unit. This unit also generates each coefficient on the fly during the execution. We first decide to recomputed the coefficients and then convert them to fixed-point. This will have an effect on computation speed, since all elements are computed and being kept into the memory. During the computation these elements can be loaded from the memory instead of being generated at each step. Besides only half of the coefficients need to be computed because of the function's periodic property. The coefficients vary between 1 and -1. This means that all intermediate values need to be converted very accurately as their fractional part contain all useful information. This is exactly the opposite case, which we already discussed for sampling unit. In this case almost all information is captured in fractional part of the data and only the first and last coefficients have their information in their integer part. In table 5.1 we shows the representation of floating-point coefficient that is used in this unit. We will also demonstrate how we convert these coefficients into the fixed-point data type to allow a fixed-point arithmetic computation instead of floating-point counterpart.

Table 5.1 shows only a part of coefficients, since all coefficients are represented in the same manner:

-0.923880	-0.529804	-0.540171	-0.545325	-0.550458
-0.555570	-0.560662	-0.565732	-0.570781	-0.575808
0.851355	0.848120	0.844854	0.841555	0.838225
0.834863	0.831470	0.828045	0.824589	0.821103
-1.000000	0.803208	-0.615232	0.780737	1.000000

Table 5.1: coefficients

As we can see almost all information is captured after decimal point. These information needs to be carefully encoded. Our aim is to use 16-bits for encoding the coefficients. Because the samples are also encoded in 16-bits, this will allow us to perform a fixed-point arithmetic by using 16-bits. We will demonstrate how we have done the conversion in this stage by using equations discussed in previous section. First we start to capture the integer part of this value. This can be shown as follows:

$$\alpha_{min} = \text{-1} \ , \ \alpha_{max} = 1$$
$$\text{QI}|\alpha_{min} \geq log_2(-\alpha_{min})+1 = log_2(1) + 1 = 1$$
$$\text{QI}|\alpha_{max} \geq log_2(\alpha_{max+1})+1 = log_2(2) + 1 = 2$$

To make sure that the calculation is performed correctly, this can be verified by doing the next calculation:

$$(2^{QI-1}) \leq \alpha \leq (2^{QI-1} \text{ -1})$$
$$(-2^{2-1}) \leq [\text{-1.000000,1.000000}] \leq (-2^{2-1} \text{ - 1})$$
$$(-2^1) \leq [\text{-1.000000,1.000000}] \leq (-2^1)$$
$$(\text{-2}) \leq [\text{-1.000000,1.000000}] \leq (1) \text{ Correct !}$$
α is bounded by minimum and maximum values of a 2-bit signed number

As it is shown in above, only two bits are required for capturing the integer part of coefficients. One bit is necessary for representing the value, since these values in integer part differs between 1 and -1. However another additional bit is required for representing the minus sign. This means that 14-bits are left, which must be used to capture the fractional part. As we can see from the coefficients table 5.1, each value contains six digits after the decimal point.

First we find out if it is possible to capture these six digits with 14bits. Equation(5.3) is used to calculate the number of necessary bits and is shown in figure 5.7.

We will start with the following number. If the calculation turns out to be correct, this means that the other coefficients can be calculated in the same way.

$$\text{X} = 0.848120\text{, this value should be assigned to } \alpha\text{, which means:}$$
$$\alpha = 0.848120 \ , \ \epsilon = 0.000001$$
$$\text{QF} = \lceil (log_2 \ (\frac{1}{0.000001})) \rceil$$
$$\text{QF} = \lceil (log_2(1000000)) \rceil = \lceil (19{,}931568569) \rceil = 20$$

As we see from the above calculation, in order to represents all digits after decimal point, 20bits are required. Since, our aim is to capture the fractional part with 14 bits, this means that we must drop a number of these digits in order to fits the digits in 14 bits. However, this approach will have a consequence on the accuracy of calculation, since these values are less accurate. In order to continue, we have decided to remove the last two digits and do the same calculations as before:

$$\text{X} = 0.8481\text{, this value should be assigned to } \alpha\text{, which means:}$$
$$\alpha = 0.8481 \ , \ \epsilon = 0.0001$$
$$\text{QF} = \lceil (log_2 \ (\frac{1}{0.0001})) \rceil$$
$$\text{QF} = \lceil (log_2(10000)) \rceil = \lceil (13{,}28771238) \rceil = 14$$

As we can see, by discarding the last two digits, we are able to representing the fractional numbers by using exactly 14 bits. At this point we have successfully converted the floating-point numbers into fixed-point numbers by using 16 bits for samples as well as coefficients. In the next section, we discuss how the original code was modified in order to modify the FFTld unit to perform fixed-point arithmetic.

5.5 Modification of FFTld Unit

In this section we will discuss how this unit is modified in order to do the computations in fixed-point arithmetic. Figure 5.9 shows the C-code, which was modified for this purpose.

This unit is responsible for doing a Radix-2 FFT/IFFT Transform and is implemented as a 1024 Points. The computation is completely performed in the memory. This unit starts by taking two complex values from the memory. These values along with the coefficients are used as the inputs to FFT Butterfly to perform a FFT calculation. After this stage the new calculated complex values will be written back into the memory. In order to finish 1024 points FFT, this process must be repeated 10 times.

```
void fft1d(int_64_complex *x, int_16_complex *w, int idir, int n) {
    int i, u, v, inca, incb, incn, j, k, ell;
    int_64_complex z;

     incn = n;
     inca = 1;

    while(incn > 1) {
        incn /= 2;
        incb = 2 * inca;
        i = 0;

        for(j=0; j<incn; j++) {
            k = 0;
            for(ell=0; ell<inca; ell++) {
                u = i + ell;
                v = u + inca;
        z.re =  ((w[k].re *  x[v].re) >> 14) - ((w[k].im * x[v].im ) >> 14) ;
        z.im =  ((w[k].re *  x[v].im) >> 14) + ((w[k].im * x[v].re ) >> 14) ;

        x[v].re = (x[u].re - z.re);
        x[v].im = (x[u].im - z.im);
        x[u].re = (x[u].re + z.re);
        x[u].im = (x[u].im + z.im);

                k += incn;
            }
            i += incb;
        }
        inca = incb;
    }


    if(idir<0) {
        for(i=0; i<n; i++) {
            x[i].re = x[i].re / n;
            x[i].im = x[i].im / n;

        }
    }

}
```

Figure 5.2: Fixed-Point FFT Butterfly in C

The FFTld unit contains four different inputs namely:

1. ***x:*** is the first input to this unit. It consists of complex samples, which are originated

from sampling unit. Although, the samples are encoded as 16 bit fixed-point, they are sign extended to 64 bits. The reason for this is because the whole computation is performed within the memory, thus this process requires more bits to correctly represent the calculated values.

2. ***w:*** is the second input and contains all necessary coefficients for this unit. These coefficients are also encoded as 16-bits fixed-point data type.
3. ***idir:*** is the third input and is used to indicate what kind of transform being performed.(FFT or IFFT).
4. ***n:*** is the last input to this unit and indicates the number of FFT points, In this case is the number of FFT points equal to 1024.

Now that we have explained the inputs to FFTld unit, the next step is to explain the calculation results from this unit. However there are some important issues, which have to be mentioned in this unit. To make sure that the results are accurate enough, we have compared these results with floating-point results. As was explained before, the conversion process and fixed-point arithmetic will affects the results. This means that the fixed-point results will not be the same as the floating-point.
Another important issue is the following code: As we can see from figure 5.10, z.re and

```
z.re =  ((w[k].re *  x[v].re) >> 14) - ((w[k].im * x[v].im ) >> 14) ;
z.im =  ((w[k].re *  x[v].im) >> 14) + ((w[k].im * x[v].re ) >> 14) ;
```

Figure 5.3: Radix-2 Fixed-Point FFT Butterfly

z.im are intermediate variables. These variables belong to a struct variable named complex and are used to represents real and imaginary part of a complex value. However, in order to make this routine suitable for a correct fixed-point data representation, after each multiplication the calculated value needs to be shifted right by 14 bits. This operation will cause to throw away 14-bits from the fixed-point coefficients and to retrieve the correct value.

The drawback of this routine is that after each calculation, the computed values will grow by 2 bits, since only 14-bits of the fractional part is discarded and the integer part has to be kept as it is. We have tried to convert the value after each calculation to 16 bits, however it seems that by doing it in this way a number of integer bits will be lost. This has a big effect on computed value, which leads to having unacceptable results.

Because of this reason, we decide to save each calculated value in 64 bits. This was required, because the new values will extend over 32 bits. In the next page we will explain how many bits are required in order to represents all values in a proper manner.

- The samples are captured in 64-bits, however only the first 16 bits contain usefull information.
- The coefficients are also captured in 16 bits, but after each calculation the new value needs to be shifted right by 14 bits.
- This process needs to be repeated for 10 times, since a radix-2 1024 point FFT requires 10 stages.
- The Number of stages can be calculated as follows: Log_2 Number of FFT point.

After 10 times the bit growth can be divided into two different cases namely:

- The best case scenario: In this case there is a bit growth of 2 bits. As it was shown in the figure 5.3, besides multiplication operations, there are also addition and subtraction operations involved. The addition operation can also cause a bit growth. During the best case scenario there is no additional bit growth and only the multiplications cause a bit growth. This lead us to have a bit growth of: 10 * 2 = 20.
  Bit growth after 10 stages is equal to : 20 + 16 = 36 bits are needed to represents calculated values.
- The worst case scenario: In this case addition also causes a bit growth and the number of the bits can be calculated as follows:
  20 + 10 + 16 = 46 bits required to represent the calculated values.

During the development we decided to use long long variable for intermediate variable as well as for input samples. long long data type is equivalent to 64-bits. Since in software we cannot define our own data types as it was required for previous case, long long (64-bit) data type is used.

In this section we explained how we did the conversion from floating-point into fixed-point data type. However the down side of this approach was that the bit growth could not be avoided and this leads to construct a FFT Transform, which requires more than 32 bits. The next step was to take this restriction into account and map this routine onto the hardware. Our aim is to build this unit as a decided processor in order to speed up the application. In the next chapter we will discuss three different approaches that we considered for this purpose and will discuss all of them as well.

6 FFT Hardware Considerations

6.1 Introduction

In this chapter we explain a number of options, which we have considered for the FFT unit. As it was mentioned in the previous chapter we have successfully converted a floating point data into fixed point. During this stage we have captured samples as well as coefficients in 16 bits fixed point. However in this chapter we will introduce three different approaches for implementing this unit as a fixed-point accelerator attached to the MOLEN pholymorphic processor. In this chapter we first start by giving information about the MOLEN architecture. Then we will discuss about the first option that we have considered for FFT design, which was to use the *Xilinx Fixed-Point FFt core*. However we found out that this unit had limitations and we have decided to use the *DWARV* toolset. However in this stage we saw that this tool was not able to satisfy our requirements, which led us to consider the third option, which was designing our custom reconfigurable FFT processor. The first two options including the MOLEN processor will be discussed in this chapter. The last option will be discussed in the next chapter.

6.2 The Molen Processor

In this section we give a brief explanation about the *MOLEN processor*. The following references are good staring points for those who need more information about this topic [6-7].

The *Molen Processor* illustrated in figure 6.1 consists of two units namely: A general purpose processor (GPP), also known as *core Processor* and a *reconfigurable processor*(RP). The RP consists of the μcode unit and the custom computing Unit (CCU). The μcode stores μinstructions that control the CCU, which is the actual HW accelerator. The aim of the *reconfigurable processor* is to be applied as a dedicate processor in order to speed up a function. This means that an arbitrary function can be implemented as a single processor in order to execute a particular operation. This is achieved through an extension of the instruction set. In addition, the MOLEN processor contains an arbiter, which is responsible for performing a partial decoding of the instructions. This process can be explained as follows: The trivial instructions sets are steered by the arbiter to the GPP. Whenever a hardware execute instruction is occurred, the reconfigurable accelerator(RP) will be activated and a branch loop will be sent to the GPP in order to keep GPP busy, while the CCU is still processing. However when the CCU is done with processing, the GPP will resume on receiving normal instructions for further processing.
They are also two other units involved during this process namely: Exchange Registers (XREGs) and shared data memory. These two units are responsible for communication between GPP and the RP. The exchange registers are used to pass function parameters and return values between GPP and RP. Larger amounts of data can be stored in the shared data memory.

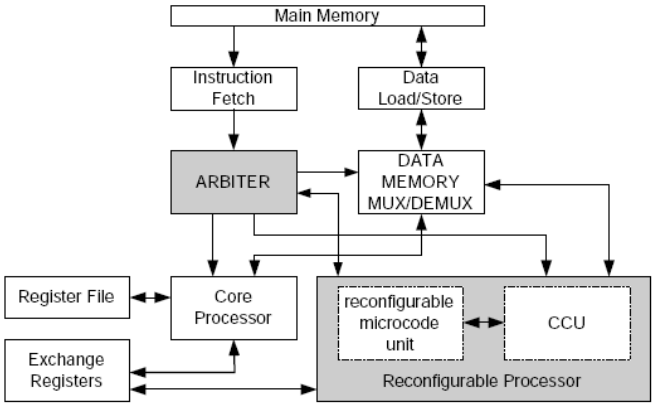

Figure 6.1: The MOLEN machine organization

6.3 Xilinx FFT Core

In this section we present the Xilinx FFT core, which we have used during this thesis. In our case we need to deploy this unit for calculating 1024-points FFT transform. The Xilinx FFT core offers a number of different architectures and also supports several arithmetic computations. The architectures can be divided as follows:

- Radix-4, Continuous I/O. This solution uses two radix-4 butterfly processing engines and offers continuous data processing by using input memory, output memory, and intermediate memory banks. The core has the ability to simultaneously perform transform calculations on the current frame of data, load input data for the next frame of data, and unload the results of the previous frame of data.
- Radix-4, Burst I/O. In this solution the FFT core uses one radix-4 butterfly processing engine and has two processes. One process is loading and/or unloading the data. The second process is calculating the transform. Data I/O and processing are not simultaneous. When the FFT is started, the data is loaded in. After a full frame has been loaded, the core will compute the FFT. When the computation has finished, the data can now be unloaded. During the calculation process, data loading and unloading cannot take place. The data loading and unloading processes can be overlapped if the data is unloaded in digit reversed order.
- Radix-2, Minimum Resources. This architecture uses one radix-2 butterfly processing engine and has burst I/O like the radix-4 version above. After a frame of data is loaded, the input data stream must halt until the transform calculation is completed. Then, the data can be unloaded. As with the Radix-4, Burst I/O architecture, data can be simultaneously loaded and unloaded if the results are presented in bit-reversed order.

In addition, three arithmetic options are available. One option is to compute the FFT using full-precision unscaled arithmetic. Also, the data can be scaled fixed-point where the user provides the scaling schedule. After scaling, superfluous LSBs can be either rounded or

truncated. The next figure 6.2 depicted the FFT Core for three arithmetic options.

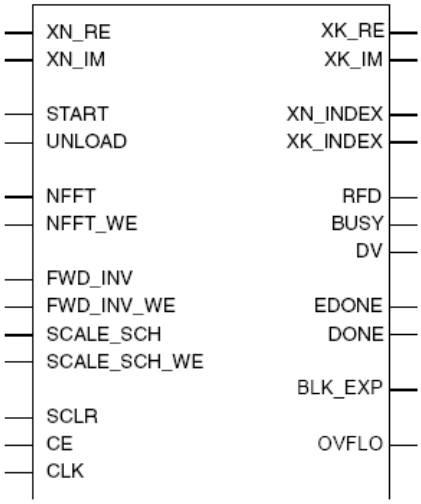

Figure 6.2: Xilinx FFT Core

We decided to use Radix-2 for our purpose. The reason for this is because the software implementation of FFT is also implemented as a Radix-2. We decide to follow the same direction as it was applied in software. The transform length is set to 1024-points as it was the case in software. The inputs are provided as 16-bits fixed-point data type. Coefficients are internally saved in the core and are also represented as 16-bit fixed-point data. We apply full-precision unscaled arithmetic, which takes into account the number of bit growth at each stage. As it was explained in the previous chapter this bit growth is almost equal to 2 bits at each stage. In order to determine the necessary bits for correct representing the outputs, the core applied the next formula:

$$\textbf{output\_data\_width} = \textbf{input\_data\_width} + log_2(\textbf{transform\_length}) + \textbf{1}$$

This approach will make sure that almost no data will be lost during the computation. However we know that doing computations in fixed-point arithmetic will always result to losing information. We decide to continue with this technique, because this provides sufficient data accuracy. In addition, as it was explained, this technique is performed in three stages. The first stage starts by loading a frame of data from a memory. In our case the size of frame is equal to 1024 numbers. Each data in frame is represented within 16 bits. The second step is responsible to process all data. During this stage the whole frame will be transformed to the frequency domain. The final stage consists of writing the transformed data back to the memory. However, to verify correctness of the FFT core we had to write a test bench according to these stages, but before discussing the test bench we have to mentioned all necessary signals which were involved during this stage.

As it is shown in the figure 6.2 the FFT unit consists of several inputs and outputs. In addition, there are a number of signals involved for activating the core during each stage, where each signal notifies a particular event for further interaction. Before discussing the test bench, we have to mention these signals that are applied as the part of FFT core for our purpose.

These signals can be divided as follows:

Input signals

- **clk:** This is the clock of the FFT core
- **ce:** Clock enable (Active High)
- **sclr:** Master reset
- **fwd_inv:** Control signal that indicates if a forward FFT or an inverse FFT is performed
- **fwd_inv_we:** Write enable for FWD_INV
- **start:** FFT start signal
- **unload:** Result unloading
- **xn_re:** Input data bus real component
- **xn_im:** Input data bus imaginary component

Output signals

- **rfd:** Ready for data (Active High) - RFD is high during the load operation.
- **xn_index:** Index of input data
- **busy:** Core activity indicator (Active High) - This signal will go high while the core is computing the transform.
- **edone:** Early done strobe (Active High) - EDONE goes high one clock cycle immediately prior to DONE going active.
- **done:** FFT complete strobe (Active High) - DONE becomes high for one clock cycle at the end of the transform calculation.
- **dv:** Data valid (Active High) - This signal is high when valid data is presented at the output.
- **xk_index:** Index of input data
- **xk_re:** Output data bus real component
- **xk_im:** Output data bus imaginary component

In order to verify the correctness of FFT core and to be sure that this core will satisfy our design requirements, a test bench is written to test the FFt core. However, to implement the test bench correctly, it is divided into a number of processes. Each process can be executed independently of each other. All these process are executed in parallel without interacting with each other. First the Active input process is shown. This process is responsible to activate the necessary input signals of the FFT core. Figure 6.3 shows code written for this purpose:

The second process is responsible for reading the input data from a file. It also has to monitor the rfd signal during this process. Upon activation of this signal, the core indicates loading the data into the core. This process must be done during this event, otherwise no data will be sent to the core. Figure 6.4 shows the code for this purpose:

```
generate_input signals: process
begin
WAIT for OFFSET;
sclr <='1';
wait for PERIOD;
sclr <='0';
fwd_inv_we<='1';
wait for PERIOD;
start<='1';
fwd_inv_we<='0';
wait for PERIOD;
start<='0';
wait;
end process;
```

Figure 6.3: Activate the input signals

```
readProcess :PROCESS
file inFile:TEXT open READ_MODE is "infile_1.txt";
VARIABLE  inLine: LINE;
variable ar_var : integer;
variable ai_var : std_logic_vector (19 DownTo 0) := (others=>'0');

BEGIN

    WAIT UNTIL (clk = '1' AND clk'EVENT);

if (not endfile(infile) and rfd = '1') then

readline(infile,inLine);
read(inLine, ar_var);

   rxn_re <= CONV_STD_LOGIC_VECTOR(ar_var,20);
   rxn_im <= ai_var;
   No_of_elements_read <= No_of_elements_read + 1;

end if;
END PROCESS readProcess;
```

Figure 6.4: Read data from FFT Core

The third process is responsible for unloading the data. Two signals have to be monitored during this process. Upon activation of these signals, we can set unload signal high to activate the FFT core for unload stage. Figure 6.5 shows how this process is done:

```
Load_Phase : process is

 begin

 wait until(clk = '1' and clk'event);

if ( busy = '1' and edone = '1') then
unload <= '1';

else
unload <='0';
end if;

end  process Load_Phase;
```

Figure 6.5: Unload data from FFT Core

The fourth process is a small process, which is responsible for counting the numbers of valid data that are represented by core. During this process dv signal is used, which indicate when a valid data is represented at the output. Figure 6.6 shows how this process is done: The final

```
OutputControl: PROCESS(clk)
BEGIN
if  dv='1' then
newValueToSave   <= TRUE;

else
newValueToSave   <= FALSE;

end if;
end process OutputControl;
```

Figure 6.6: Valid data count

process is responsible for writting the results into the files. The results consists of Real and imaginary data's. The core uses dv signal to indicate this phase. Figure 6.7 shows this process.

```
    file outFile:TEXT open WRITE_MODE is "outfile1.txt";
    VARIABLE  outLine  : LINE;
    file outFile1:TEXT open WRITE_MODE is "outfile2.txt";
    VARIABLE  outLine1  : LINE;
  BEGIN
    WAIT UNTIL clk = '1' AND clk'EVENT;
    if ( dv = '1' ) then
        hWRITE(outLine,result_re,right, 27);
        WRITELINE(outFile, outLine);
        hWRITE(outLine1,result_im, right, 27);
        WRITELINE(outFile1, outLine1);
    END IF;
  END PROCESS WriteProcess ;
```

Figure 6.7: write process

6.3.1 Xilinx FFT Core Simulation

In this section the simulation of Xilinx core is shown. We emphasize on the three stages, described in the previous section. Figure 6.8 shows how each stage is performed. Also all necessary signals are shown during each stage. Our aim is to show that the simulation is exactly executed as it was described in Xilinx datasheet. During load process the rfd signal stay high, which indicate that the core is busy with reading a frame. Each frame consists of 1024 complex numbers. After loading the whole frame, the rfd signal goes low, which indicates the completion of read process. The following stage is the computation stage, where the busy signal stays high until the core is done with all computations. After this stage edone, done and unload signal goes high for one cycle. Finally, dv signal will be activated during write process to write the calculated frame back to the memory.

Other important signals are xn_index and xk_index. These signals can be applied as indices for selecting data from memory during read and write back process. All of these processes are shown in the following figure 6.8.

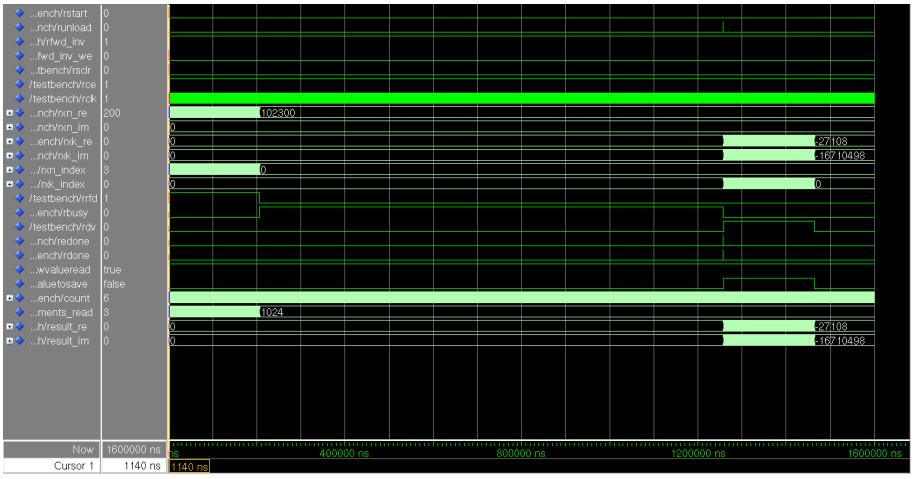

Figure 6.8: Xilinx FFT Core Simulation

As it is shown in figure 6.8 all of these processes are exactly performed as it was required for each stage. However we haven't discussed the results of this process. This is very important, since we really like to know how accurate this core is. Besides, we are interested to know how accurate are the fixed-point results and how big the differences in relation to floating-point data are. In the next we will discuss this subject in more details.

6.3.2 Results of FFT core

In this section will shows the results, which we were obtained during the simulation. Our aim was to evaluate the Xilinx FFT core with the software version to find out how suitable is this core for our purpose. We are especially interested in how accurate is this FFT core, since the software version is implemented in floating-point data. We first show the real part of computed numbers in term of precision. Precision refers to how close values agree with the software ones. Then we show the second computed result in term of accuracy. Accuracy refers to how closely a value agrees with its true value. The following formula (6.1) show how to calculate the relative error for each calculated number against its floating-point version. The relative error present the absolute error relative to the exact value.

- $\epsilon rel = \frac{\|\tilde{x}-x\|}{\|x\|}$ **(6.1)**
- $\tilde{x}$ = fixed-point number
- x = floating-point number

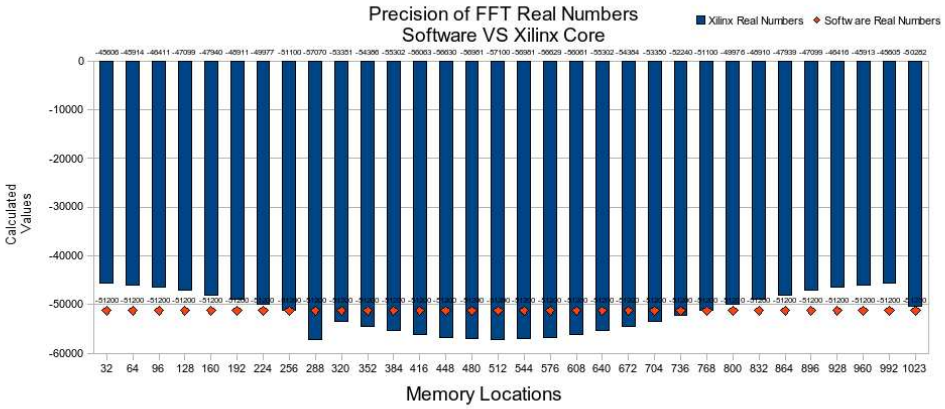

Figure 6.9: Xilinx Real numbers(Precision).

Figure 6.9 shows the calculated numbers in terms of precision. The calculated numbers from the Xilinx FFT core have a different representation in contrast to the software version. There is a gap between the numbers, which indicate the differences for each calculated value and its real representation.

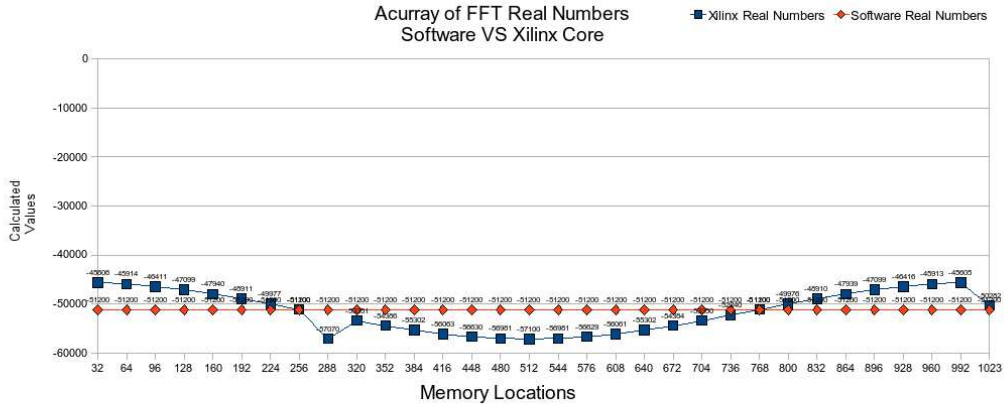

Figure 6.10: Xilinx Real numbers(Accuracy).

Figure 6.10 shows the calculated numbers in terms of accuracy. The calculated numbers from the Xilinx FFT core have a different representation in contrast to the software version. There is also a gap between the numbers, which indicate the differences for each calculated value and its real representation. Figure 6.11 shows the absolute error between the calculated

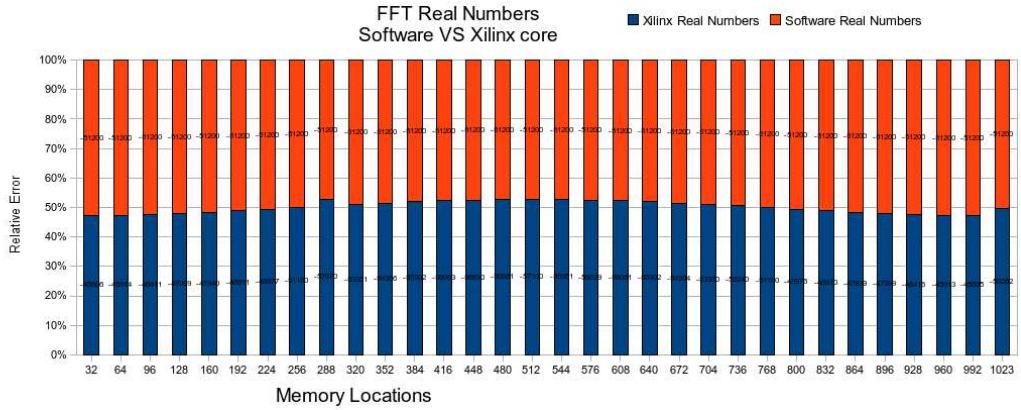

Figure 6.11: Xilinx Real numbers(Relative Error).

numbers of the Xilinx core and software version. As it is shown, there is an overlap between the values, which indicate the difference between these versions in terms of relative error. Each line shows the error between two calculated values. The accuracy between a floating-point and a fixed-point number refers to the difference between their integer and fractional part. In contrast, the precision calculation considers only the integer part of a floating-point number during the calculation with a fixed-point number. The relative error is the percentage difference between the the floating-point and fixed-point numbers. Figure 6.12 shows how precise are the calculated

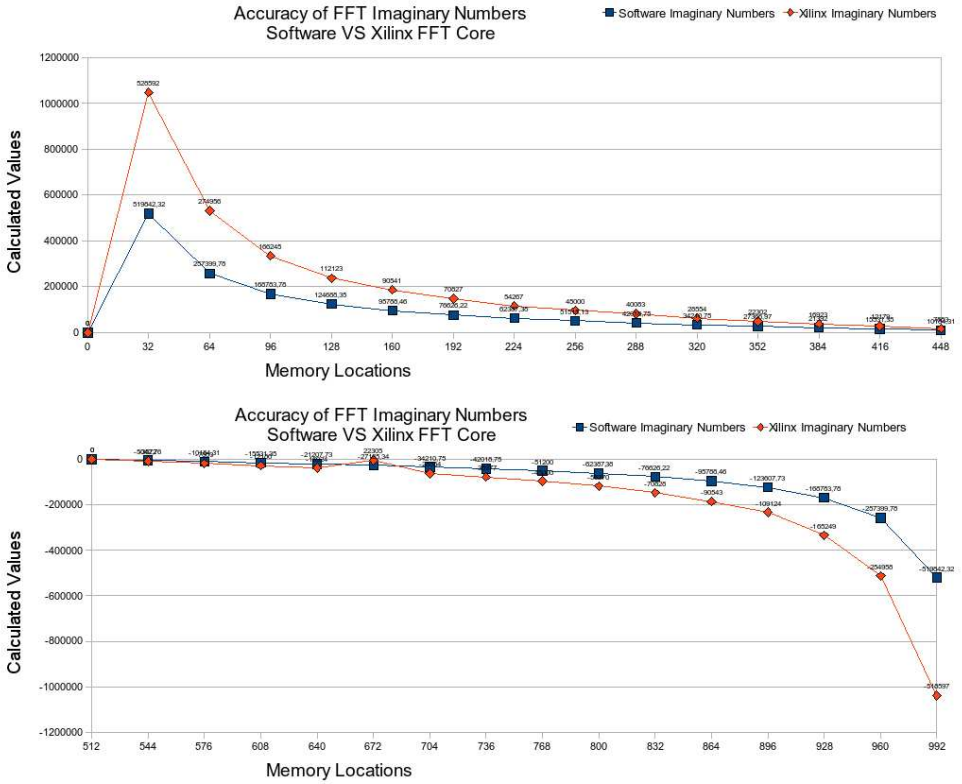

Figure 6.12: Xilinx Imaginary numbers(Accuracy).

imaginary numbers of the Xilinx core in respect with software version. As it is shown, there is a gap between the values, which indicate the difference between these versions in terms of precision. These graph also shows the relations as it was described for real part. Besides, as it is shown the core is not really sufficient to represent the true values of calculated numbers comparing to floating-point numbers. This can be seen by analyzing these figures, which depicts the huge differences for these calculated numbers.

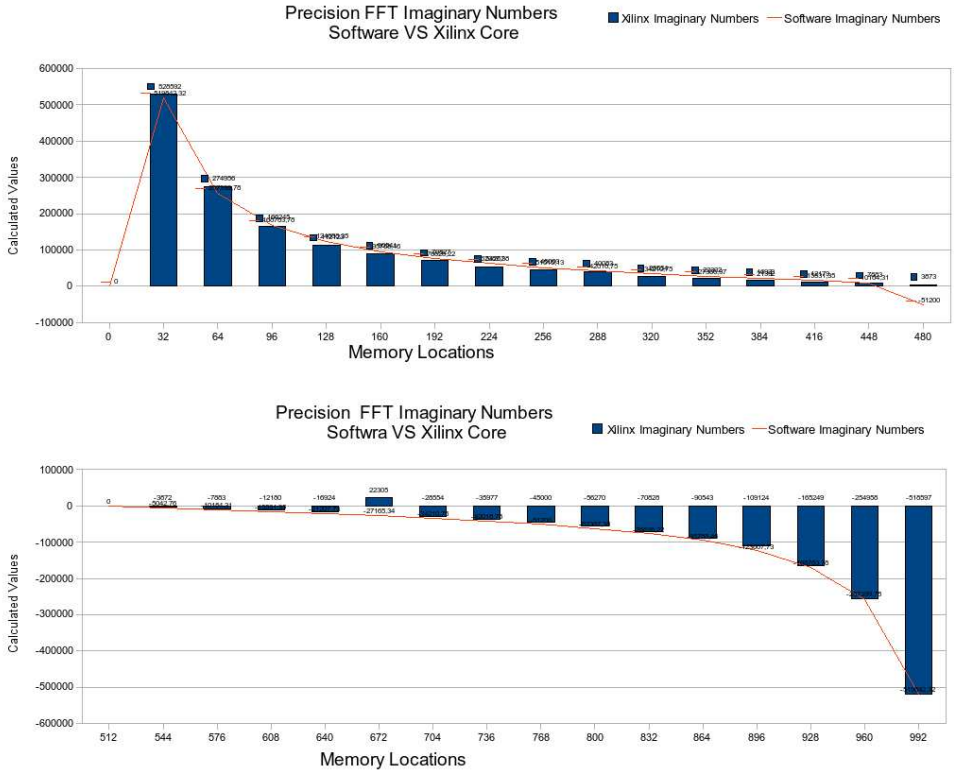

Figure 6.13: Xilinx Imaginary numbers(Precision).

Figure 6.13 shows how accurate are the calculated imaginary values of Xilinx core in respect with software version. As it is shown, there is a gap between the values, which indicate the difference between these versions in terms of accuracy. Figure 6.14 shows the relative error between the calculated imaginary values of Xilinx core and software version. As it is shown, there is an overlap between the values, which indicate the difference between these versions in terms of error numbers.

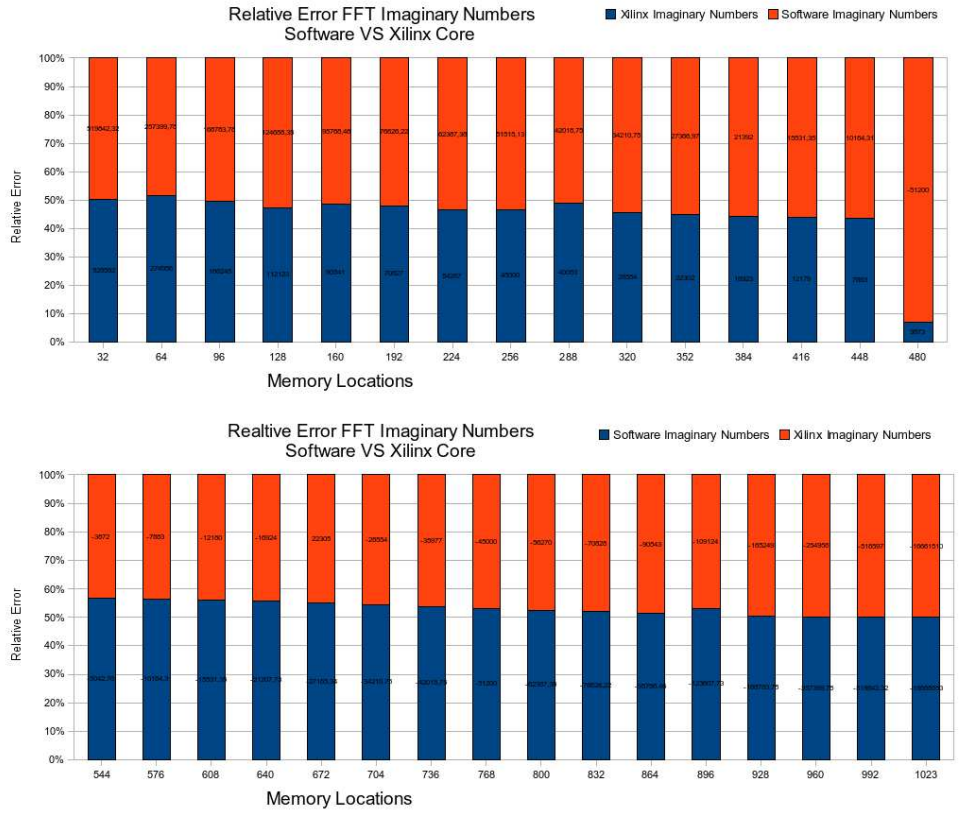

Figure 6.14: Xilinx Imaginary numbers(Relative Error).

The above results show the precision, accuracy and the absolute error of Xilinx FFT core in relation with software version. The FFT starts with Bit reversion function and after the calculation the results are stored as normal order. However, we have seen that the core provides results with huge variation in respect with the floating-point version. Although the core use 31 bits to represent the calculated value, we have seen that there is a huge difference between the calculated value and floating-point version. At this point we have decided to consider the second option, which will be discussed in the following section.

6.4 DWARV tool chain

In this section we will discuss about the DWARV tool set, which we have considered as the second option during this thesis. DWARV is a semi automatic toolset, which is developed to be applied with MOLEN reconfigurable processor [8,9]. During this thesis we have decided to use this tool, since it offers a great possibility for our case. The major advantage of this tool is that this allows the developers to describe the hardware unit in C software language instead of VHDL.
This tool is able to accept a C code, which contains describing of a Hardware unit. Subsequently, the tool will generate a file, describes hardware unit in VHDL. DWARV is also able to automatically connect the generated unit with CCU using the MOLEN interface with the GPP. However there are some restrictions. One restriction is, it cannot support the struct type in C. There are also number of other restrictions, which are listed in the following documents[10,11]. In order to use this tool for our purpose we had to modify the code according to DWARV user guide document.

We have decided to use this tool to implement the FFT core including the CCU interface in order to employ it with MOLEN processor. Moreover, since the software based FFT core consists of three separate units, we had to merge these units into just one. The software code has to be implemented exactly as it was described in DWARV user guide. Another important property of DWARV is the pragma annotations. By placing a pragma next to the desired function, this pragma will notify the compiler to generate that function as a desired Hardware unit. The parameters of the function have to be placed in a separate file. During the application execution the program checks this file to fetch the parameters.

The BRAM memory is used by MOLEN during the application execution. However to employ BRAM during this stage the memory elements has to be placed in a separate file. DWARV will transform these data to BRAM coefficients and the hardware unit will have access to these elements during computations. After this stage is completed the results will be written to a separate file.

The next code shows how we modified the software in order to make it suitable for DWARV to generates the FFT core.

```
    #pragma to_dfg /* annotate the function for HW generation */
 static void perm_fft(int *s_1,int *s_2, int *re, int *im,short m) {
   unsigned short n, i,r , ir, k;
    int temp;

   unsigned short i_1, u, v, inca, incb, incn, incn_1, j, k_1, ell;
    int z_re,z_im;

    n = 1 << m;
  for(i=0; i<n; i++){
ir=0;
 for(k=1; k<m; k++) {
    ir = ir | ((i >> (k-1)) & 1) << (m-k);

}

    if(ir > i) {

           temp = s_1[ir];
           s_1[ir] = s_1[i];
           s_1[i] = temp;
```

```
    temp = s_2[ir];
            s_2[ir] = s_2[i];
            s_2[i] = temp;

        }
    }
inca = 1;

      for(incn = n; incn>1; ) {
        incn /= 2;

        incb = 2 * inca;
        i_1 = 0;
        for(j=0; j<incn; j++) {
            k_1 = 0;
            for(ell=0; ell<inca; ell++) {
                u = i_1 + ell;
                v = u + inca;

                z_re = ((re[k_1] * s_1[v])/16384) -  ((im[k_1] * s_2[v])/16384) ;
                z_im = ((re[k_1] * s_2[v])/16384) +  ((im[k_1] * s_1[v])/16384);

                s_1[v] = (s_1[u] - z_re);
                s_2[v] = (s_2[u] - z_im);
                s_1[u] = (s_1[u] + z_re);
                s_2[u] = (s_2[u] + z_im);
                k_1 += incn;
            }
            i_1 += incb;
        }
        inca = incb;
  }
}
```

As it was mentioned, in order to use DWARV tool set the code has to be modified. The parameters are changed from a complex struct into two separate arrays. The first array contains the Real part of the samples and the second array the imaginary part of the samples. This is also applied for coefficients. In addition, BitRev and swap functions form the first part of FFT unit. These units are responsible for doing the necessary memory operations, after which is the last part responsible to perform the butterfly calculations. We have used the above code as an input to DWARV tool. The tool was able to generate VHDL code for the FFT core. However the next step was to simulate the generated VHDL code in order to assess the correctness of the code. During this stage we have noticed that the results were not correct as it was expected. The reason for this was that DWARV toolset was not able to generate a precise VHDL code, because the tool was restricted to 32 bits integer. However our unit requires more than 32-bits. We have spent some time to debug the unit and try to locate the problem. But this was very hard because the code was very difficult to understand. Besides, the data path could not support more than 32-bits and this was the reason that we have decided not to continue with this tool. Since the tool was not capable to generates correct code for our purpose. We have decided to start designing our custom FFT core, presented in chapter 7.

7 Custom FFT Design

7.1 Introduction

The aim of this chapter is to explain about the FFT core that has been developed during this thesis. Since we have decided to develop a custom FFT, our first step was to study 9.1 suitevarious already purposed hardware FFT architectures. Based on our studies, we start to design a FFT core in VHDL using the ISE Xilinx tool. Finally the complete architecture of a memory-based FFT processor is implemented.

7.2 Implementation of the FFT processor

In this thesis, we have decided to implement a Fast Fourier Transform (FFT) processor with radix-4 algorithm. An in-place buffering strategy [14-15-20] is chosen. The design is based on a four-point DFTs of 1024-points instead of radix-2 as it was the case in software. The aim of this design is to perform a FFT transform, which is implemented in hardware as a co-processor next to the MOLEN reconfigurable processor. The FFT computation is based on fixed-point arithmetic in contrast to floating-point data type, which was the case in WFS application. The aim of this is to speed up the computation, since fixed-point arithmetic requires less clock cycles instead of floating-point counterpart.

At the top level we partition the FFT processor into three submodules. Data Input Process, FFT and Data Out Process as shown in figure 7.1 [29].

7.3 Memory-Based FFT Architecture

The first step of processing starts with the data input process. During this phase samples are read from the BRAM and stored into the main memory in order to be prepared for the next stage. In the second phase, the data is read from the main memory and the FFT is computed. Results are written back into the same memory. During the computation, the data type is a complex number and each number consists of a real and imaginary part. Each part of the complex number is equal to 32-bits and is represented as a fixed-point type.

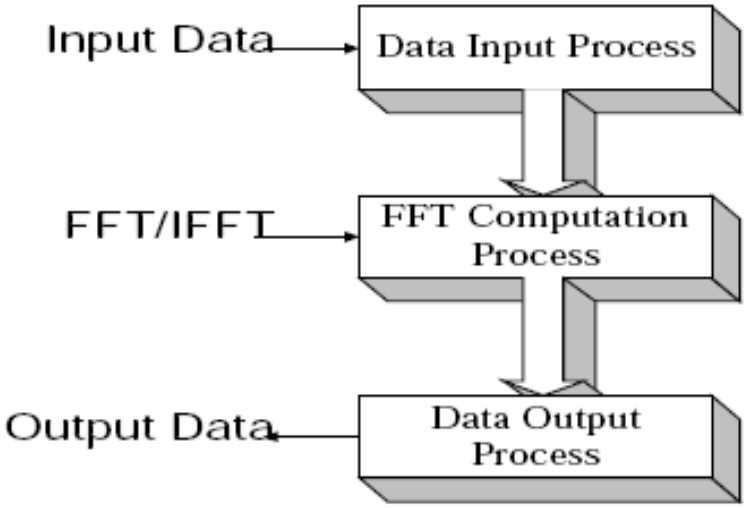

Figure 7.1: Top level description of the operation of the FFT processors.

7.3.1 Block Diagram of the FFT Processor

Figure 7.2 shows the FFT design of our custom processor. There are two additional units, which are also involved in this design namely: Input unit and output unit. These two units are required in order to read and store the data according to the MOLEN memory structure and FFT architecture. These two units including our FFT Processor will be discussed later in more details. Moreover, there is also another unit involved during this design, which is not shown in this picture. This unit is implements the CCU interface between our FFT processor and MOLEN. The communication between the MOLEN processor and the FFT processor is executed through this interface. In the following subsections we will give more details about the implementation of this unit.

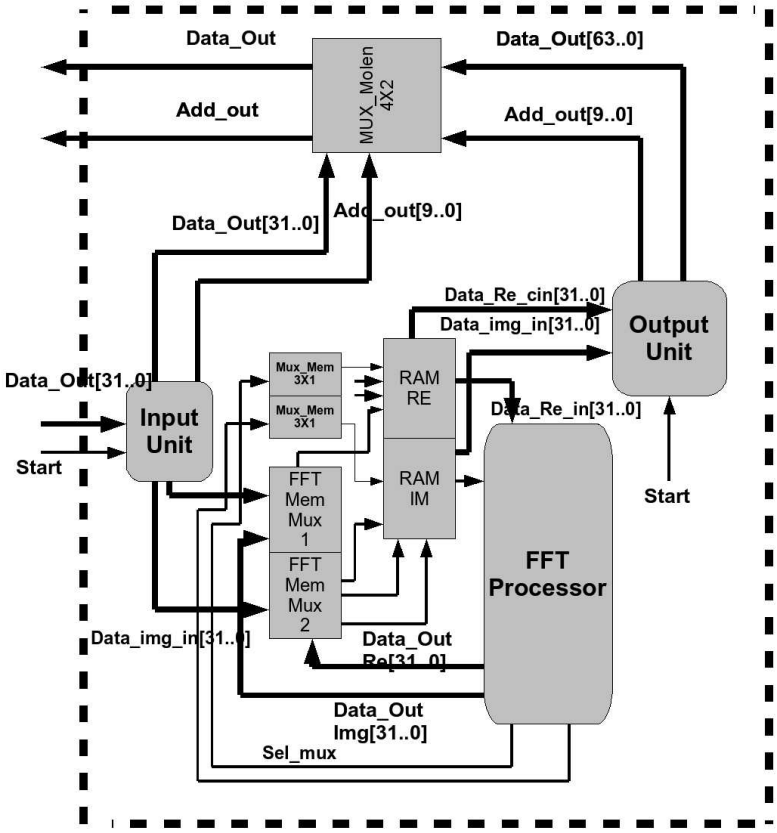

Figure 7.2: General overview of Block diagram of the FFT processor.

7.3.2 Input Unit

The input Unit is the first unit of the FFT processor. This unit is specially designed to cooperate with the MOLEN shared memory (BRAM). Since BRAM structure requires the data to be stored as 64 bits, each location can store two 32 bits of words. This corresponds to having two numbers at one location in BRAM. However during data reading our intention is to read data from the BRAM and stores them at main memory of the FFT processor. The data in BRAM are stored according to the following definition:

- From 0 until 511 locations are considered to be used for storing 1024 numbers of Real type.
- From 512 until 1023 location are considered to be used for storing 1024 numbers of Imaginary type.

However, since the FFT processor is implemented as a 32 bit processor, the input-unit is first responsible to read 64-bits data from BRAM, which correspond to two separate 32 bit values. This 64 bit data reside in one location, which can be accessed by the unit according to a calculated index. Finally, all retrieved data are stored to the main memory. Figure 7.3 shows how this unit is constructed:

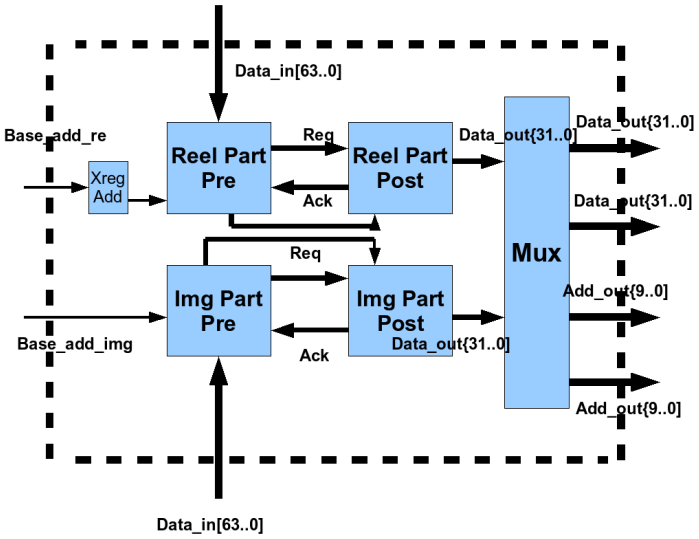

Figure 7.3: Input-Unit of the FFT Processor.

As it is shown in figure 7.3, the input-unit consists of two pre and post for real and imaginary part. The pre unit is responsible to activate the BRAM in order to read two numbers represented as one 64-bits word and forward it to the next subunit (post-unit). The pre-unit contains a counter, which is used to generate indices for the data that are read from the BRAM. In addition, this counter uses a base address, which is originated from the MOLEN. This is necessary to identify the first location of the first number in order to generate indices based on this location. Additionally, there are some signals involved for communicating between the pre and post units. These signals are necessary in order to exchange the data according to an asynchronous protocol.

Each data from the BRAM is read by pre unit and will be sent to the post unit when the pre unit receives a signal from the post unit. This signal (ack) is generated after the post unit is finished by storing 64-bits to two different locations of the main memory. Each of these locations contains 32-bits of data. Furthermore, the post unit is also implemented as a counter-based design and is responsible for generating indices for main memory in order to store data correctly. We have decided to choose normal order for bitrev indexing during this stage, which means that the data at this level will be read and stored in a normal order.

The output unit is also implemented according to the same principle similarly to the Input-units, expect that the former will first start reading the values from the main memory after FFT computation is done. The pre-unit in this process is responsible for reading two values from the main memory and send them to the post-unit. An asynchronous protocol is implemented for the communication between each unit. The pre-unit is activated when the post-unit is finished by concatenating two 32 bits as a one 64 bits and store it to the BRAM. However, the post-unit is does not store the data normal order instead it uses the Bitrev principle to place each data correctly into the calculated locations. Another problem was how to correctly store two 32-bits data from the main memory to BRAM. For instance, the data from the locations 0 and 1 cannot be stored as a 64-bit in location 0 of BRAM, since this will leads to incorrect data storage. The reason for this is because in Bitrev operation 0 will correspond to 0 locations, however 1 will be corresponded to the location 511 in BRAM. To solve this problem correctly we have decided to use the first 32 bits for the real data and second part of the data for imaginary data. This means

that after Bitrev function location 0 of the BRAM will contains the Real part and Imaginary part of the data. This leads to have a complex number stored at each location of BRAM. These units will have a major influence on performance of the whole design, since, all operations are executed sequentially. In the next subsection we will discuss more details about the components of the FFT core.

7.4 Components of the FFT processor

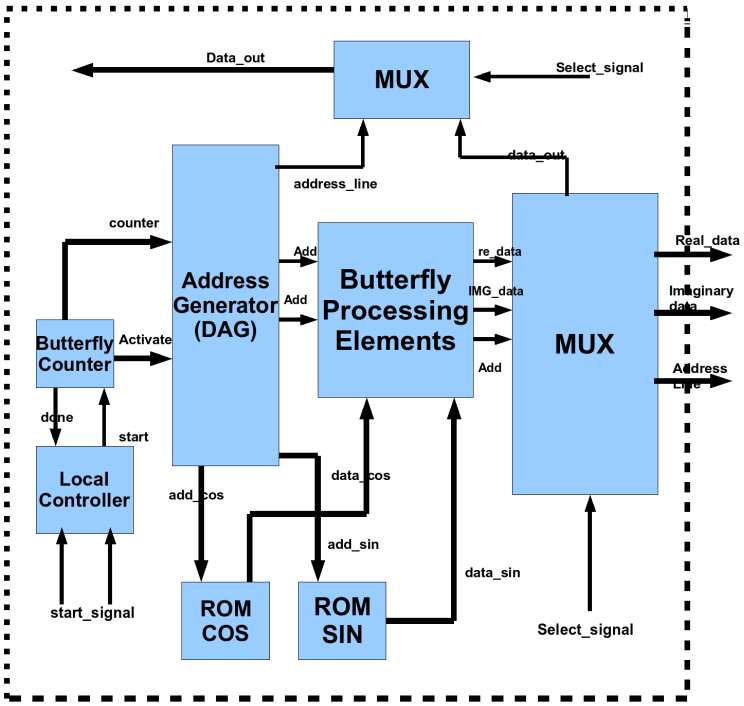

Figure 7.4: The Core of the FFT Processor.

As it is shown figure 7.4 the FFT processor consist of a number of sub units. Each of these will be discussed in the following subsections.

7.4.1 Data Address Generator Unit (DAG)

The aim of the DAG is generate correct address for the RAM as well as for ROM memory units. It also keeps track of which stage is being executed. Furthermore, this unit is activated by the FFT local-controller. The latter is also responsible to activate the butterfly counter, which then provides data to the address generator with all the necessary addresses and the stage number. This unit is only activated during FFT computation, since input and out units are responsible of data reading and writing from and to the BRAM.

The address generation for Radix-4 is based on the following equation shown in the next page. However this concept is modified from bitwise (radix-2) to digitwise operation(radix-4). The four addresses required by radix-4 (radix-2^2) butterfly processing element are defined as [s,

t, u, v] and can be computed by the following Equations[22].

$$s = \left[i_{\log_4 N-2}\, i_{\log_4 N-3} \cdots i_{\log_4 N-1-k}\ \mathbf{0}\ i_{\log_4 N-2-k}\, i_{\log_4 N-3-k} \cdots i_1\, i_0 \right]_4$$

$$t = \left[i_{\log_4 N-2}\, i_{\log_4 N-3} \cdots i_{\log_4 N-1-k}\ \mathbf{1}\ i_{\log_4 N-2-k}\, i_{\log_4 N-3-k} \cdots i_1\, i_0 \right]_4$$

$$u = \left[i_{\log_4 N-2}\, i_{\log_4 N-3} \cdots i_{\log_4 N-1-k}\ \mathbf{2}\ i_{\log_4 N-2-k}\, i_{\log_4 N-3-k} \cdots i_1\, i_0 \right]_4$$

$$v = \left[i_{\log_4 N-2}\, i_{\log_4 N-3} \cdots i_{\log_4 N-1-k}\ \mathbf{3}\ i_{\log_4 N-2-k}\, i_{\log_4 N-3-k} \cdots i_1\, i_0 \right]_4$$

In above equation i = [$i_{log_4^{N-4}} i_{log_4^{N-3}}$ $\ldots i_2 i_1 i_0$] corresponds to the butterfly numbers and k is the stage number. The data addresses for butterfly processing element in direct processing order are listed in Table 7.1.

stage number	Butterfly 0	Butterfly 1	Butterfly 2	Butterfly 3	
stage 0	(0, 4, 8, 12)	(1, 5, 9, 13)	(2, 6, 10, 14)	(3, 7, 11,15)	Real Parts
stage 1	(0, 1, 2, 3)	(4, 5, 6, 7)	(8, 9, 10, 11)	(12,13,14,15)	
stage 0	(0, 4, 8, 12)	(1, 5, 9, 13)	(2, 6, 10, 14)	(3, 7, 11, 15)	Imaginary Parts
stage 1	(0, 1, 2, 3)	(4, 5, 6, 7)	(8, 9, 10, 11)	(12,13,14,15)	

Table 7.1: Data address for butterfly PE in direct processing order.

To verify the above table and also give more explanation of how to apply this equation in order to calculate the desired address, the following example is used to demonstrate the calculations in more details:

For calculating the desired value required by the 2^{re} butterfly of k stage(1^{st} stage) in a 16-point radix-4 FFT processor, the calculation can be performed as follows:

Equation: i = 2, k = 0

2^{rd} butterfly = $[02]_4$, log_4 N - 1 - k = 2 - 1 - 0 = 1

Applying the above calculation into previous equation we obtain the following address :

Sr = $[02]_4 = 2_{10}$

tr = $[12]_4 = 6_{10}$

ur = $[22]_4 = 10_{10}$

vr = $[32]_4 = 14_{10}$

If we compare the calculated address indices with the table 7.1 we can observe that these values are correctly calculated. Each of these values can be used for real as well as for the imaginary part, since we have decided to use two separate memories. For real and imaginary parts. The reason for this is because by dividing the memory storage into two parts, the data

address generator will be simpler, since only one index has to be generated and will be used for real as well as imaginary parts. The hardware block diagram of read data address generator for our FFT processor is shown in figure 7.5:

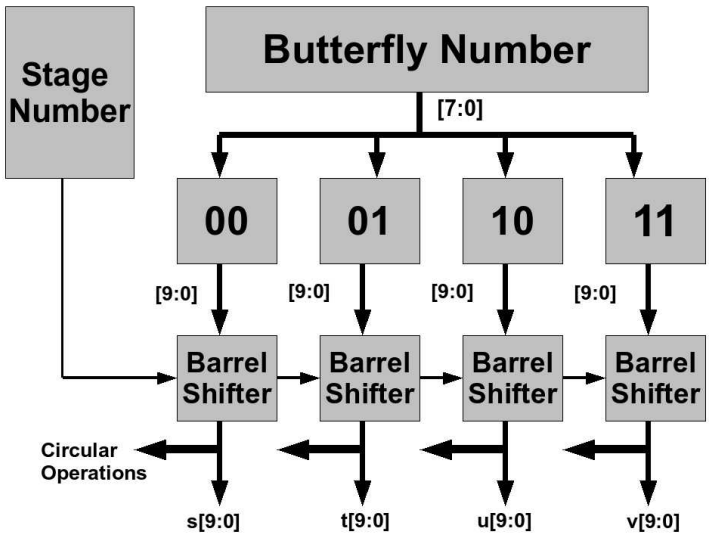

Figure 7.5: Block diagram of a data address generator.

As we can see from figure 7.5, the first unit is responsible to extend the butterfly numbers in order to have enough bits for selecting all data from the memory. There are also 4 Barrel-shifters involved. Each shifter performs a circular shifting according to the stage number, since different addresses are required during each stage.

7.4.2 FFT Write Back Address

This unit is responsible to keep track of address indices for storing the data into the RAM. However, we can use the addresses, which are generated during the Read Address stage. The reason for this is because we utilize the in-place algorithm. This leads us not to generate additional addresses in this stage. Instead we make sure that indices from the Read Address unit are registered and reused. When the butterfly finishes with calculations all calculated values are stored back into the memory according to registered indices. The FFT core will signal the Write back unit in order to provide the values with a correct address. However, this signal is depended on how many cycles are required by the butterfly to complete a calculation.

7.4.3 Butterfly Counter

The butterfly counter is responsible for keeping track of which butterfly is being used in a particular stage. This counter is only applied during the FFT computation, since the input and output units have their counters. The butterfly counter is also responsible to increment the stage number after the last butterfly iteration has occurred. This counter will be activated by the FFT local controller, however after this stage the butterfly counter will be incremented by

a signal, which originates from FFT core. The reason for this is to make sure that the current butterfly is done with all computations before sending the new value to next one. A butterfly requires 7 cycles to process data before activating the butterfly counter.

7.4.4 ROM Address Generator

The coefficient (twiddle factor) plays an important role in the FFT process. This unit contains real and imaginary data coefficients. However, since these values are already known, they can be pre computed and be stored in a ROM memory. In addition, there is a ROM address generator required, which is used to get the coefficient data. In our design we used two ROM memories to store cosine and sine data. This unit will be activated during the FFT computation and is responsible to provide the required values to the multiplier at the same time. To demonstrate the design of a simple ROM address generator we will use an example, which shows how to generate a correct address for the ROM table to read the desired coefficients.

This example takes a 64-point radix-4 FFT algorithm, which consist of three stages and the twiddle factors needed in every stage are listed as follows:

First Stage:	$W_{64}^{n}\ ,\ W_{64}^{2n}\ ,\ W_{64}^{3n}$	n = 0 ... 15.
Second Stage:	$W_{64}^{4n}\ ,\ W_{64}^{8n}\ ,\ W_{64}^{12n}$	n = 0 ... 3.
Third Stage:	$W_{64}^{0}\ ,\ W_{64}^{0}\ ,\ W_{64}^{0}$	n = 0 ... 0.

From the above equation, we can observe that the coefficients, which are required in the second stage, can be obtained from the twiddle factors in the first stage. Therefore, by only calculating the coefficients for the first stage we can use these coefficients during every stage. The block diagram in figure 7.7 shows the hardware implementation of the ROM Address Generator. The hardware consists of two units namely: a barrel-shifter and a mathematical unit.
The barrel shifter is responsible to generate the partition number DFTs-point. However, this can be different at each stage. To determine these numbers, the barrel shifter uses the butterfly number and based on the stage number the unit generates the desired partition number. This is done by using arithmetic shifting. The second unit is responsible to take numbers from the previous unit and based on stage number to calculate all desired indices. However the bits are extended from 8-bits to 10-bits in this stage. This was necessary in order to select the values from the ROM, since this unit contains 1024 values and an addressing width of 10-bits.

7.4.5 Butterfly Processing Element

The Butterfly processing element (BPE) shows in figure 7.7 consists of several multiplier and adder units. However, except from the first butterfly, the rest contain also a shift-register, located after the multiplier. The aim of this register is to discard the 14-bits after multiplication. However, the first butterfly doesn't involve any shift register and also uses no coefficients. The reason for this is the property of twiddle factor, since it must always calculate the DFT of four values, where the coefficients are equal to 1. We have decided to not include the unit with multipliers and shift-registers, since these are not required. However, this also leads this unit to be finished earlier than the other units, since fewer cycles are required during this stage. We have decided to delay the output of this unit in order to retrieve all calculated data from all butterflies at the same time.

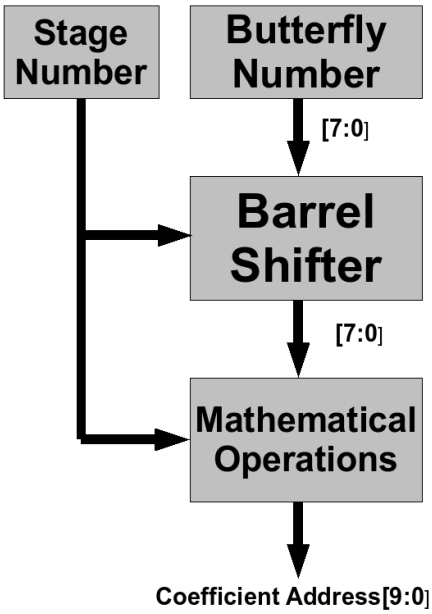

Figure 7.6: Radix2^2 coefficient index generator.

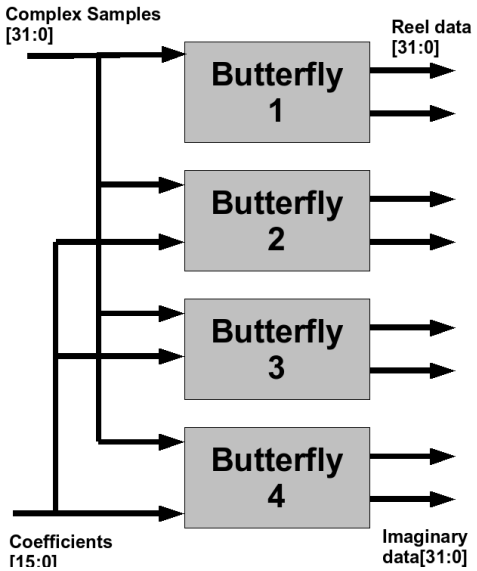

Figure 7.7: Block diagram of the butterfly processing element.

7.4.6 Processing Elements

The processing element is based on the $Radix - 2^2$ processing element. Each butterfly takes 4-points of input data values A,B,C and D and outputs X0,X1,X2 and X3 as shown in figure 7.9:

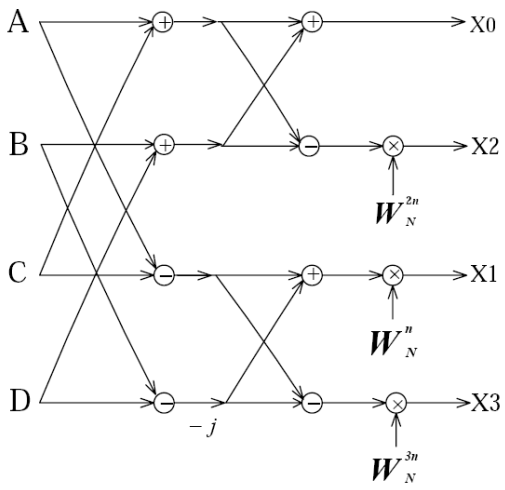

Figure 7.8: $Radix2^2$ Processing element.

In order to implement each butterfly for $Radix2^2$, we have to rewrite the $Radix-2^2$ equation as follows:

- RE[X0] = A_re + B_re + C_re + D_re
- Im[X0] = A_im + B_im + C_im + D_im
- Re[X2] =(A_re - B_re + C_re - D_re) $cos2\theta$ + (A_im - B_im + C_im - D_im) $sin2\theta$
- Im[X2] =(A_im - B_im + C_im - D_im) $cos2\theta$ - (A_re - B_re + C_re - D_re) $sin2\theta$
- Re[X1] = (A_re + B_im - C_re - D_im) $cos\theta$ + (A_im - B_re - C_im + D_re) $sin\theta$
- Im[X1] = (A_im - B_re - C_im + D_re) $cos\theta$ - (A_re + B_im - C_re - D_im) $sin\theta$
- Re[X3] = (A_re - B_im - C_re + D_im) $cos3\theta$ + (A_im + B_re - C_im - D_re) $sin3\theta$
- Im[X3] = (A_im + B_re - C_im - D_re) $cos3\theta$ - (A_re - B_im - C_re + D_im) $sin3\theta$

The hardware implementation of the butterfly is shown in figure 7.10. The butterflies first starts by taking 4 values from the memory and compute it. After this stage the results are written back to the memory. The computation during this stage is equal to 7 cycles. The butterfly unit consists of 4 multipliers, 8 adders and 4 shift-registers. It involves also a number of registers, which are used to buffer the input. Multiplexers are also applied, which are used during writing the calculated values back into the main memory. After the butterfly is finished with computation, the controller will reset the bufferfly and start the next computation. The

computation is based on 32-bits fixed-point data type.

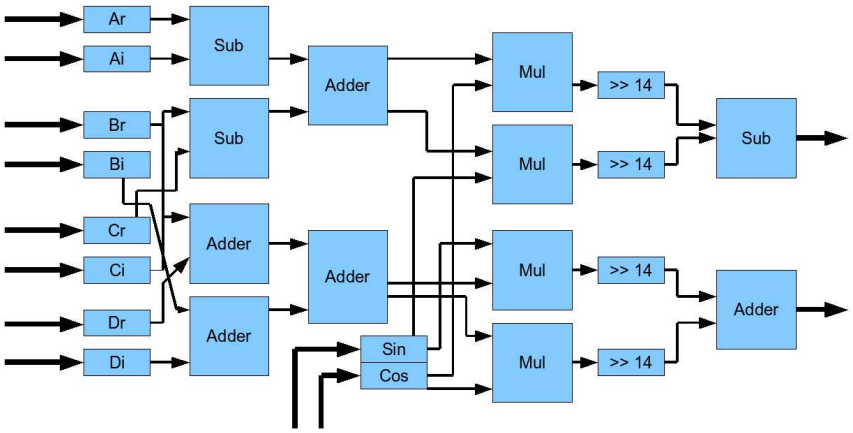

Figure 7.9: One of $Radix2^2$ butterfly processing element.

we have designed four butterfly shown in figure 7.9, each element according to figure 7.8. These four units are placed in parallel as shown in figure 7.7. The only difference among them is the first unit, which has no multiplication and shift-registers. The architecture of the other units are exactly identical.

7.4.7 Controller Unit

The FFT controller shows in figure 7.10, consists of 5 states and it is implemented as a finite state machine. Upon its activation, the unit starts by reading the data from the BRAM and write them to the main memory. During this stage the input process unit is activated to accomplish this task. The unit will first start by reading the real numbers from the BRAM and store them to the main memory according to their generated indices. The process is also repeated for imaginary numbers. After this stage is completed the unit has to send a done signal to the controller. Hereafter, the controller has to activate the FFT core in order to perform the FFT computation. As soon as the FFT core finishes all calculations, it acknowledges the controller unit and the new values are stored to the main memory. Finally, the controller has to activate the out process unit to transfer the data from the main memory back to the BRAM. The process will be started by reading a real and imaginary data from the main memory and store them to the BRAM according to the calculated indices. The process will be repeated for all real and imaginary data, which must be transfered from the main memory to the BRAM. After the execution of this process the unit will signal the controller for its completion and send the controller back to its first state.

7.4.8 ROM

In our design we have used two ROM memory blocks. The first one is used to store the cos values of the twiddle factor and the second one is used to store sin values of the twiddle factor. The size of these memory blocks is equal to 1024, thus a 10-bit address is used.

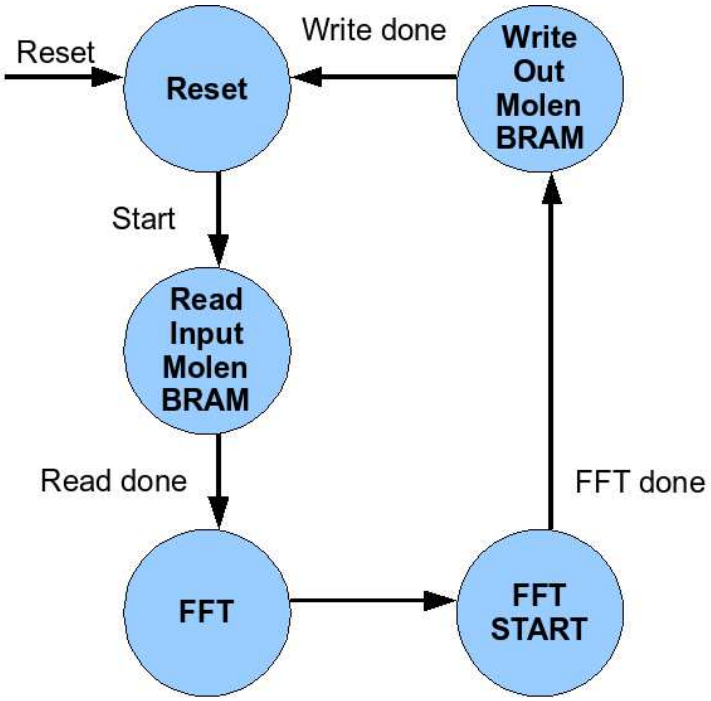

Figure 7.10: State Diagram for a FFT processor controller.

7.4.9 RAM

In our design we use two RAM memory blocks. The first one is used to keep the cos value and the second one is used to keep the sine value. By using two separate memories we can read and write the data into the memory at the same time while FFT unit computes the butterfly. During the input process data are first written into the RAM from the shared BRAM. During the FFT computation process, four numbers are calculated and written back into the same location in the RAM. During the output process the data are read from the RAM and stored back into the BRAM according to the Bit Reversal principle.

7.4.10 CCU Controller Unit

The MOLEN controller is responsible for exchanging parameters between the GPP and the RP and also to control the communication between the MOLEN processor and our FFT processor. It is implemented as a finite state machine and consist of 11 states as illustrated in figure 7.11.

The GPP sends two parameters to the FFT processor in order to identify the base address of real and imaginary data. The FFT processor takes these addresses and starts reading data from the desired location, in order to fill the main memory. We have used two exchange registers from MOLEN in order to keep the base addresses. Figure 7.11 shows how the state machine is implemented.

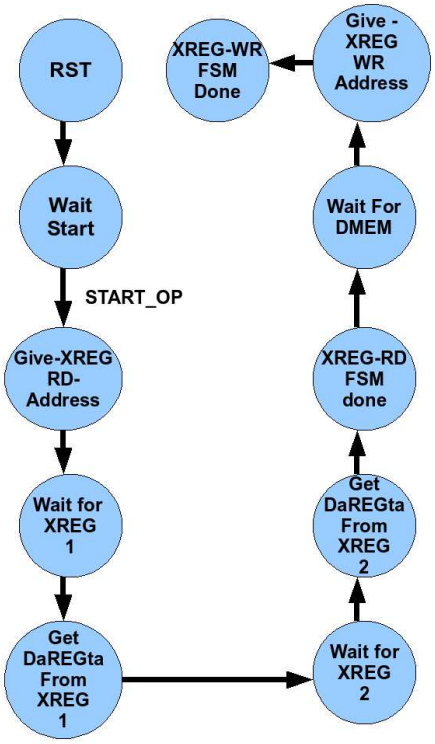

Figure 7.11: State Diagram for a MOLEN controller.

After the controller receives a start signal, this unit will be activated for exchanging the parameters between the GPP and the FFT core during the input as well as output process. This unit will read the first parameter from the XREG unit. It will wait until it receives the data from the XREG unit. Then the parameter will be read and saved into a register, since the same parameter will be used during the output process. The second stage is also similar to the first stage, because during this process two parameters are required. After saving the two required parameters, this unit must wait until the core is finished with the FFT calculation. Upon activation of this unit, it start by returning the values to the GPP through the XREG unit. The controller has to use the same base address, which were used during the input process to read the the data from the BRAM into the main memory. These address can be used to store the data back into the BRAM during the output process.

Design Evaluation and Experimental Results

8

This chapter consists of 4 sections. In section 8.1 we explain the result of the synthesis process, which was performed in order to synthesize the design in the hardware. In addition, during this stage the program generates a report, which contain the hardware utilization about the FFT design. The design is synthesized by using the XST synthesis tool. We show the results for the FFT core as well as for the core including MOLEN interface. In section 8.2 we present the simulation results. The calculated numbers are compared with the software version as it was the case for the Xilinx core. The results illustrate the different between the calculated numbers in terms of accuracy, precision and relative error. In section 8.3 we report various FFT designs found in the literature. However, all of the designs are based on a pipelined architecture, while different technologies are chosen during hardware implementation. We provide a brief analysis of these designs and compare them against our proposed hardware FFT core. In section 8.4 we describe the new improved execution time for the WFS application, which is estimated by using the Amdahl's Law. The overall speedup of the application is calculated based on the execution time during the hardware simulation and the utilization time of FFT according to the software profile results. Finally, in section 8.4 we present a design comparison between our design and a pipeline based design in term of hardware consumption.

8.1 Synthesis of the FFT Processor

Table 8.1 presents the results for the FFT core, while table 8.2 contains the results for the FFT core including MOLEN interface. The designs are synthesized and implemented using the Xilinx ISE 9.2. ModelSim 6.0 is used to simulate and verify the designs. During the synthesis stage the XC2VP30 device of Virtex 2Pro family of Xilinxs FPGA is selected. The synthesis tool has allocated the following resources for our custom FFT core: 4474 slice registers (17%), 2825 Flip Flops (5%) and 7826 lookup table (15%). We have performed the same approach for the FFT design including the MOLEN interface, which is presented in table 8.2. The synthesis tool has allocated the following resources for this unit: 5124 slice registers (20%), 3430 Flip Flops (6%) and 9123 lookup table (18%). Table 8.1 and 8.2 show the result of this process for each design in more details.

Logic Utilization	Used	Available	Utilization
Number of Slices	4474	25280	17%
Number of Slice Flip Flops	2825	50580	5%
Number of 4 input LUTs	7826	50580	15%
Number of bounded IOBs	68	576	11%
Number of FIFO16/RAMB 16s	6	232	2%
Number of GCLKs	1	32	3%

Table 8.1: Synthesis results of FFT Core

Logic Utilization	Used	Available	Utilization
Number of Slices	5124	25280	20%
Number of Slice Flip Flops	3430	50580	6%
Number of 4 input LUTs	9123	50580	18%
Number of bounded IOBs	113	576	19%
Number of FIFO16/RAMB 16s	10	232	4%
Number of GCLKs	1	32	3%

Table 8.2: Synthesis results of FFT Core including MOLEN Interface

8.2 Simulation of the FFT Processor

In this section we present the results, that are obtained during simulating the FFT core. ModelSim is used during the simulation stage to verify and measure the execution time of the FFT design. This step was required to obtain an accurate execution time for the FFT core. The other requirement was to verify the correctness of the calculated numbers. In order to verify the results during the simulation, the same input numbers are applied as it was the case with the Xilinx FFT core. The only difference is that during the simulation of our custom FFT core, we start with normal order to read the data from the shared memory. Whenever, the core is finished with all calculations, the results are stored back to the shared memory according to bit reversal. The reason that DIF is chosen instead of DIT, the latter is the case in the software version, to simplify the process for the software application. Since the FFT function is implemented as a reconfigurable hardware accelerator for the MOLEN processor, the software has to provide the data to the core through the shared memory and writesthe numbers as 64-bits. However as it was explained in chapter 7, each location contain two words of real or imaginary numbers. The numbers can be read from the shared memory and being used during the calculations. Whenever the core completes all calculations, the original data locations can not be employed to store new data, since bit reversal is required to store the data according to the correct order. To solve this restriction in a proper way we have decided to stores the real and imaginary numbers at the same location according to bit reversal. This implies that the software application can easily access a location of the shared memory to read a complex number instead of a real or imaginary number as it was the case before the FFT execution.

During the simulation stage, we have first measured the execution time for the radix-2 floating-point in software, on a PC with Core2 2.2 GHZ processor. During this process a function is implemented, which measures the execution time of the radix-2 FFT on a PC. ***gettimeofday*** function is called twice during the FFT execution to measure the elapsed time between the start and completion of the FFT. The execution time is reported in μSec. The measured execution time for the FFT function on the PC was equal to 1554 μSec to perform a radix-2 FFT of 1024-points. The same approach is also performed in Modelsim to measure the required execution time for our FFT design. The measured time for the hardware design was equal to 164 μSec. Also, the execution time for the core including the MOLEN interface is measured in Modelsim, which was equal to 308 μSec. Having results for both software and hardware implementation, the following formula is applied to find out how much faster the custom FFT core can execute a task comparing to the software version. The following calculation show how this process is calculated by using the formula (8.1):

- **Speedup** = SN = $\frac{T_s}{T_p}$ **(8.1)**
- **Speedup** = FFT core = $\frac{1554}{164} = 9,47560X$ **(8.2)**
- **Speedup** = FFT including MOLEN Interface = $\frac{1554}{308} = 5.045X$ **(8.3)**

The speed up for the custom FFT design is calculated according to 8.2. As it is shown, the core is almost 9X faster then the radix-2 floating-point version implemented in software. The same process is also repeated for the core including the MOLEN interface as it is shown in 8.3. The calculated speedup for this part is 5X faster then the PC version. The result for this part is limited only to the above calculations, since we can not provide other input numbers during the simulation. The reason is because our custom core can only execute 1024-points FFT as it is the case in the application. We have experienced that the WFS application requires to execute only 1024-points FFT, each time this function is executed.

The second step during this section is to evaluate the calculated numbers in term of accuracy, precision and relative error. As it was the case in chapter 6.3.1, we have verified the Xilinx core design with the software floating-point version. In this section we compare the calculated numbers of our custom design against the software floating-point version. The calculated numbers are represented in three different graphs as it is shown in figure 8.1 and 8.2. Figure 8.1 show the result for real part, while figure 8.2 show the result for imaginary part. Formula (8.4) shows how to calculate the relative error for each calculated number against its floating-point version. The relative error present the absolute error relative to the exact value.

- $\epsilon rel = \frac{\|\tilde{x}-x\|}{\|x\|}$ **(8.4)**
- $\tilde{x}$ = fixed-point number
- x = floating-point number

Formula (8.4) is used to calculate the relative error for the calculated numbers of real as well as the imaginary part. We have calculated the difference between the numbers of the custom core and the floating-point version. The results led to have a small difference of $2.8*10^{-8}$ for the real part and $1.1*10^{-5}$ for the imaginary part. This shows that our custom FFT core is much more accurate than Xilinx core, which was discussed in chapter 6. In addition, we have experienced that by using 32-bits data, this approach allows to achieve a consistent execution.

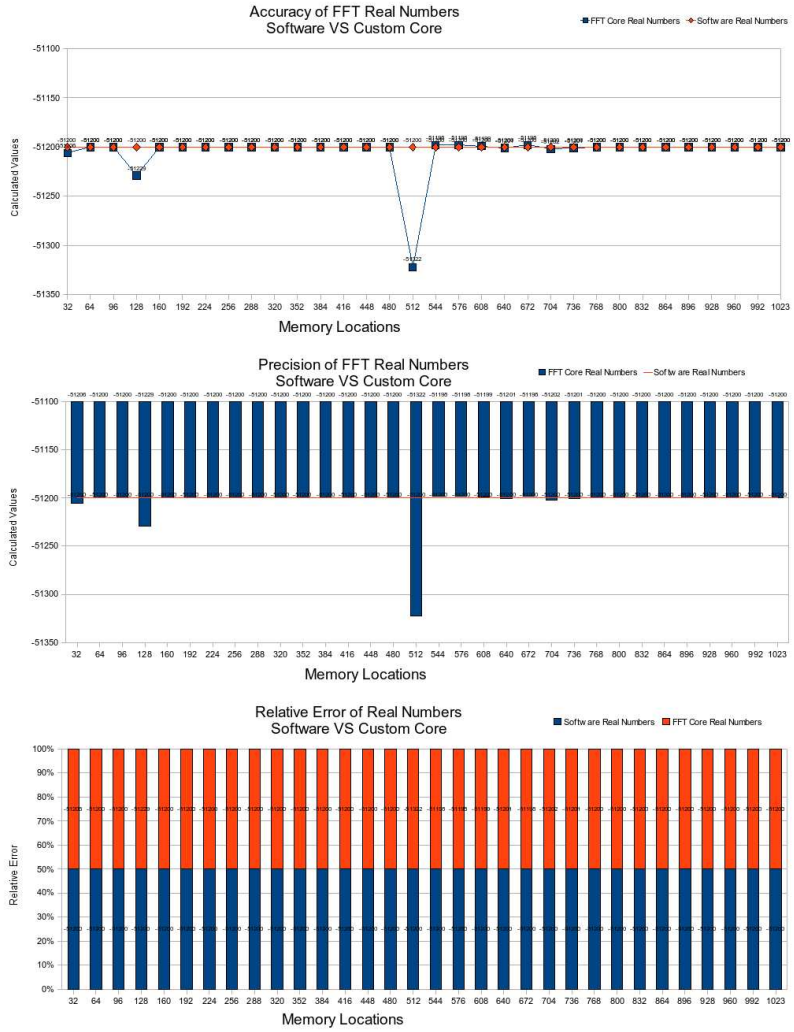

Figure 8.1: FFT Results of Real Numbers.

Figures 8.1 shows the results, that are obtained from our custom design for the real part. The first figure represents the calculated numbers in the terms of accuracy. This graph shows how closely are the calculated numbers related to their true value. During this case, for the floating-point numbers their integer as well as the fractional parts are considered during the calculation. The calculation is based on the accuracy of each number by taking the difference from fixed-point number against the floating-point number. The second graph represents the precision between the fixed-point numbers and the floating-point numbers. In this case for the floating-point numbers only the interger part is considered. The result is based on the difference between the fixed-point numbers and floating-point numbers, while only the integer part of the floating-point number is considered. The final graph shows the percentage difference between a floating-point number and a fixed-point number. In this case the graph led us to conclude that our results are correctly represented for the calculated fixed-point numbers in relation with

floating-point numbers. As it is presented in third graph of figures 8.1 and 8.2, there is almost no difference between the calculated fixed-point numbers and the floating-point numbers. However, in case of Xilinx core in chapter 6 we have seen that the core was not sufficient enough to represent the calculated numbers(6.11 and 6.14) as it is shown in figure 8.1 and 8.2. By comparing the results of figure 8.1 and 8.2 with the Xilinx core results in chapter 6 we can conclude that our custom FFT design has a higher consistent behavior than the Xilinx core. Figures 8.1 and 8.2 show how closely the numbers are related to each other, in constract to the Xilinx core, which shown that for the same computations the core will add almost 5% to 8% error to the calculated results.

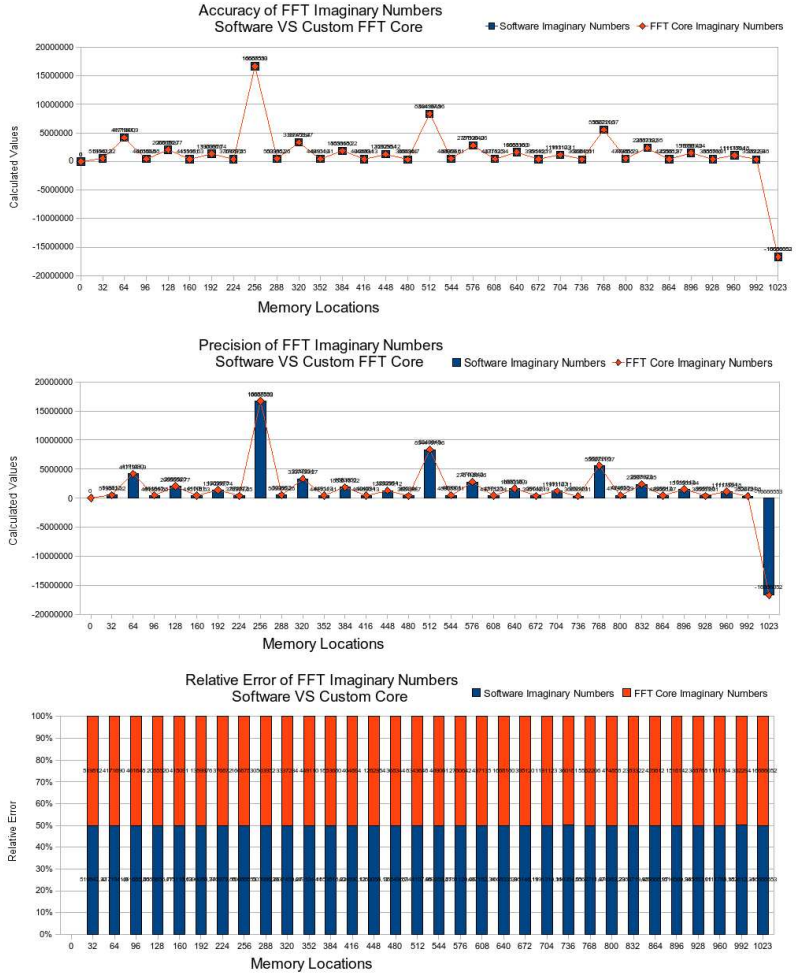

Figure 8.2: FFT Results of Imaginary Numbers.

8.3 Related Work

In this section, we present the results of several FFT designs. All designs are implemented as fixed-point processor. However, all of these designs employ a pipelined architecture, while our design is a memory based one. Moreover, all of these processors are also optimized for various purposes. The following table 8.3 shows an overview of several fixed-point FFT processors:

Processor	Chip Type	Number of Point	Data Width(bit)	Execution Time(*micros*)	Frequency (MHz)
Xilinx FFT Core	FPGA	1024	16	41.5	100
Altera FFT Core	FPGA	4096	16	262	94
Stanford, Imagine	DSP	1024	-	20.6	180
S. M. Currie FFT	ASIC	1024	16	11	100
J.C. Kuo NTU	ASIC	1024	16	40	80
Chu Chao	FPGA	1024	16	10.1	127
Jess Garca1	FPGA	256	16	0.123	350
H.L.Lin	ASIC	64	16	-	20
Y.W.Lin	ASIC	128	10	-	110
Y.H.Lee	ASIC	2048	16	-	75
T.Y.Sung	ASIC	8192	16	-	150
Y.W.Lin	ASIC	8192	11	-	20
A. Saeed	FPGA	256	16	0.135	465
Our Design	FPGA	1024	32	164	125

Table 8.3: Performance comparison with other fixed-point FFT processors [30] [31] [32][33]

As it is shown in table 8.3, there are several fixed-point FFT processors, where each one considers a certain technology for its implementation. The ASIC technology is mainly considered due to the reduced power consumption and improved speed. All these ASICs designs are based on an optimized version of the pipelined architecture. Furthermore, many optimizations are applied to reduce power and area consumption. To improve the execution time, some special arithmetic unit are involved, that can execute a radix-2 FFT in only one clock cycle.

For the FPGAs, the same considerations as it was the case for ASIC designs, are investigated. The only different is that during the FPGA design, the designer do not have to spend a lot of time with designing the chip. This is also depicted in these papers, which are using the FPGA as an intermediate step before considering a ASIC technology. This means that the FPGAs are used as a tool for a fast verification of the design. Another important point is that none of these authors report the accuracy, which is also an important topic specially for fixed-point processors. Thus it is really difficult for us to compare our design with these available chips, since our design needs to be optimized in several ways before evaluate our core with these designs. All of these tasks are discussed in the further work.

8.4 Performance evaluation

In this section we calculate performance improvement, which is obtained by utilizing our custom FFT core instead of the software version. **Amdahl's Law** can be used to calculate the performance gain that can be obtained by improving some portion of the application. It states that the performance improvement to be gained from using some faster mode of execution is limited by the fraction of the time the faster mode can be used. **Amdahl's Law** defines the *speedup* that can be gained by using a particular feature. It states that we can make an enhancement to a machine that will improve performance when it is used. Speed up is the ratio:

- ***Speedup*** $= \frac{Performance\ for\ entire\ task\ using\ the\ enhancement\ when\ possible}{Performance\ for\ entire\ task\ witout\ using\ the\ enhancement}$ **(8.5)**
- ***Speedup*** $= \frac{execution\ time\ for\ entire\ task\ without\ using\ the\ enhancement}{execution\ time\ for\ entire\ task\ when\ possible}$ **(8.6)**

Speedup describes us how much faster a task will run using the machine with the enhancement as opposed to the original machine. **Amdahl's Law** gives us a quick way to find the speedup from some enhancement, which depends on two factors:

1. *The fraction of the computation time in the original machine that can be converted to take advantage of the enhancement.*
2. *The improvement gained by the enhanced execution mode; that is, how much faster the task would run if the enhanced were used for the entire program.*

The execution time using the original machine with the enhanced mode will be the time spent using the unenhanced portion of the machine plus the time spent using the enhancement:

- ***Executiontime***HW = ExecutiontimeOld X ((1 - Fractionalenhanced) + $\frac{Fractionalenhanced}{Speedupenhanced}$) **(8.7)**

The overall speedup is the ratio of the execution time:

- ***Speedup***overall=$\frac{Executiontimeold}{Executiontimenew} = \frac{1}{(1 - fractionalenhanced + \frac{fractionalenhanced}{Speedupenhanced})}$

  (8.8)

In order to evaluate the overall speedup gained by incorporating our custom FFT core into the application, the formulas (8.7 and 8.8) can be applied to calculate the new execution time for these unit and overall speed up for the whole application:

By analyzing the table 3.1 from the chapter 3, there are three operations shown in this table 3.1 where each one has a contribution during the FFT execution. These functions are:

1. *fftld* execution time during the application is equal to **36.50%, 107.42 Sec**
2. *bitrev* execution time during the application is equal to **8.51%, 25,04 Sec**
3. *perm* execution time during the application is equal to **3.03% , 9,00 Sec**

All of these operations are also implemented in the hardware, this means that during executing the application these three operations are executed in the hardware. The Speedupenhanced of our custom FFT core is presented in section 8.2. Based on these calculations, the new execution time and speedup of the application can be calculated as follows:

HW execution time:

- 107.42(fftld) + 25.04(bitrev) + 9.00(perm) = 141.46 ***Sec***, the time is measured based on the FFT and IFFT execution, however since we have only implemented the FFT unit in hardware we have to divide this number by 2, thus 70.73 **Sec**. This will give us the correct time duration of the FFT unit during executing the application. Since we consider only the time that FFT is running, then then enhanced ratio will be 1. Using the (8.7) formula, we can calculate the FFT execution in hardware as follows:

- (141.46 / 2) x ((1 - 1) + $\frac{1}{9.4756}$) = 70.73 x ($\frac{1}{9.4756}$) = 7.46

- According to the calculated numbers during the execution, the FFT unit will be improved by 10X. However, this was expected, since the calculated values in section 8.2 already shown us the speedup for the hardware unit against the software version.

The next calculation presents the overall speedup according to formula (8.8) and shows the new execution percentage time of the entire application:

- ***Fraction enhanced***= *fraction fftld(36.5) + fraction bitrev(8.51) + fraction perm(3.03)* = 48.04% , again as this was the case in the previous calculation, we have to divide this number by 2 in order to obtain the correct value for the FFT unit, thus 24.02%.
- Speedup enhanced = 9,47560
- ***Overall Speedup*** = $\frac{1}{0.7598 + \frac{0.2402}{9.4756}} = \frac{1}{0.02534} =$ **1.273643**

According to calculated number, we have obtained an application speedup of 27%. This implies that by executing only the FFT unit of the application in hardware, the application is executed ***27%*** faster as it was the case for the software version. The following calculation will show us the improved time for executing the application based on the speedup from the previous calculation. The new execution time is based on formula (8.7) and is calculated as follows:

Overall ExecutionTime HW: = 294.3 (1 - 0.2402) + (1 - 0.2404) + $\frac{0.2402}{9.4756}$ = 231.06 ***Sec***.

According to the new execution time, if we execute the application again by using the FFT hardware unit, the execution time will be improved to 231.06***Sec*** instead of 294.3***Sec*** . This number indicate a speedup of almost **27%** on the whole application.

8.5 Design comparison

In this section we present a comparison between our design and pipeline architecture in terms of hardware consumption. Table 8.4 shows a estimation between our FFT design and a pipeline based FFT processor:

Algorithm		Radix-2	Radix-2	Radix-4
Architecture		SDF	MDC	Our design(Memory Based)
Hardware utilization	Multiplier	50%	50%	100%
	PEs	50%	50%	100%
Hardware requirement	Multiplier	8	8	3
	PE	10	10	1
	Memory(registers)	1023	1534	8

Table 8.4: The hardware utilization and requirement for different FFT architecture

For the pipline based design we have estimated the hardware usage based on a radix-2 algorithm. In the case of radix-4 the hardware usage will increase, since more hardware resources are required for processing more data at the same time. As it is shown in table 8.4, our design requires less hardware resources and use one PE during the entire processing stage and is utilized 100% during the computation. In case of pipline based architecture 10 individual PEs are required and they are only utilized for 50% during the computation. Besides, pipeline based design requires huge number of registers for input and output buffering, while in our case only 8 additional registers are applied during the computation.

9 Conclusions

9.1 Summary

In this thesis, a design of a 32-bits fixed-point Radix-4 FFT processor with memory-based architecture has been proposed. The design is able to perform a 1024-point radix-4 FFT based on fixed-point data format to accelerate the execution of the Wave Field Synthesis acoustic algorithm. The design is also simulated and synthesized on XC2VP30 Virtex 2Pro and applied as a reconfigurable hardware.

In chapter 2, we started by giving a brief introduction about the Wave Field Synthesis acoustic algorithm. We have explained how this techniques divides its data into two kinds of information. We have also explained two convolution techniques in frequency-domain, which are applied in digital signal processing specially dealing with the long real-time signals. In chapter 3, we have profiled the application to identify its critical time consuming kernels. We have used Gprof for this purpose and compiled the application with GCC compiler under Linux operating system. After this step, we have decided to analyze the results to identify about these kernels clearly. We have discovered that FFT is the most time consuming kernel of the application. At this point, the next step was to study the FFT theory in order to understand the aim of this process. In chapter 4, we have presented the theoretical background on the FFT. We have discussed several classifications of the FFT algorithm. We have also explained two different FFT hardware architectures. In chapter 5, we have performed a floating-point to fixed-point conversion. This was necessary in order to find out whether this processing can be done accurately in fixed-point arithmetic before considering any FFT hardware design. We were able to represent the FFT kernel in 16-bit fixed-point. Two bits are used to represent the integer part and the other 14 bits are used to represent fractional part. In chapter 6, we have considered three options to implement an FFT processor. The first option was to use the Xilinx FFT core. However, we found out that this core was not sufficiently accurate to continue with. The second option, was to use the DWARV tool set. Also this tool was not sufficient, since al datapaths were internally limited to only 32-bits. The last option was to implement a custom FFT core. In chapter 7, we present a 32-bits fixed-point radix-4 FFT processor with memory-based architecture. The FFT processor has been implemented by using VHDL and ModelSim for simulation. The final circuit was synthesized and implemented with Xilinx ISE 9.2. The implementation of this design led to achieve a clock frequency of 125 MHz. Furthermore, the results from fixed-point arithmetic were compared to the floating-point counterpart, which led to having a small difference of $2.8*10^{-8}$ for the real part and $1.1*10^{-5}$ for the imaginary part. We have also compared our result to the Xilinx core and based on the results could conclude that the Xilinx core for the same computations adds almost 5% to 8% error to the calculated numbers. Finally, we have calculated the overall speedup for the application, which was equal to 27%.

9.2 Objective Coverage

The main thesis goals defined in section 1.2 were:

1. Profile the WFS application and identify the time consuming kernels.
2. Converting the time consuming kernel from floating-point to fixed-point format.
3. Considering the available hardware option to design a reconfigurable unit.
4. Design a fixed-point custom FFT core as a reconfigurable processor.
5. Integrating the custom FFT core in order to be applied as a RP unit of the MOLEN processor.

- Objective 1: This objective was fully achieved. During this stage we have written a shell script to cause the application to be executed in a number of times. Gprof was applied during this stage to obtain an overview of the time consuming kernels. The next step was to reorder these kernels in order to find out the relation between them. This led us to identify the FFT unit as the most time consuming unit of this application(section 3.2).
- Objective 2: This objective was fully achieved. During this stage we have performed a coversion of floating-point into fixed-point format. The FFT unit is captured in 16-bits instead of floating-point. We were able to convert the samples as well as the coefficients from the floating-point into the fixed-point format. For the samples we have used 16-bits to capture all samples, which are originated from the ADC. To represent the coefficients 14-bits are used to capture the fractional part and 2-bits to represent the integer part(section 5.4).
- Objective 3: This objective was fully achieved. During this stage we have considered three different options for implementing a FFT hardware processor. The first option was to consider the Xilinx FFT core, however based on our results we found out that this unit was not sufficient enough for our purpose(section 6.3.2). Our next step was to consider the DWARV toolset, which is a semi-automatic tool for generating a VHDL code based on c-code. However this tool is only able to generate a fixed-point up to 32-bits. We have found out that this tool was also not sufficient for our purpose, since during the computation the fixed-point arithmetic requires up to 64-bits for internal number representation(section 6.4).
- Objective 4: This objective was fully achieved. During this stage we have decided to design a custom FFT core for our purpose. After analyzing the code further and study the FFT theory we have decided to design a radix-4 FFT core. The reason for this approach was that radix-4 can execute the computation much faster then the radix-2. Besides this approach allows us to restrict the bitgrowth, since less stages are required during the computation stage(chapter 7).
- Objective 5: This objective was fully achieved. During this stage our design is integrated into the CCU Controller Unit. We have design a CCU-interface, which will be activated for exchanging the parameters between the GPP and the FFT core during the input as well as output process(section 7.4.10).

9.3 Further work

The following proposes can be considered as the further work for this thesis.

- The core is only capable to execute 1024 points FFT. This can be modified in order to make the core generic instead of only performing 1024 point FFT. However the other options should be a power of four, since this core is designed to execute a radix-4 algorithm.
- Another modification would be to design a variable address generator. This option would make the core capable by switching between a radix-4 or a radix-2 algorithm. This option would make this core suitable to execute a split algorithm, which usually consists of a combination between radix-2 and radix-4.
- Another modification is to extend or modify this core in order to have a floating-point arithmetic. Only the datapath is required to be modified, while the address generator can be kept without any modification.
- Another modification is to reduce the datapath from 32-bits to 16-bits. This approach will significantly speedup the core. However to perform this task some special arithmetic units need to be designed.
- The final option is to add an additional functionality into the core by extending this core to also performing an IFFT operation. This will leads the core to performing a FFT as well as an IFFT transform.

References
10

1. WAVE FIELD SYNTHESIS A PROMISING SPATIAL AUDIO RENDERING CONCEPT Gnther Theile Institut fr Rundfunktechnik (IRT) Munich, Germany

2. Torger, A. & Farina, A., Real-time partitioned convolution for ambiophonics surround sound, IEEE workshop on applications of signal processing 2001.

3. Fixed-Point Arithmetic: An Introduction Randy Yates

4. Reconfigurable Accelerator for WFS-Based 3D-Audio Dimitris Theodoropoulos, Georgi Kuzmanov, Georgi Gaydadjiev

5. Wave Field Synthesis for 3D Audio: Architectural Prospectives Dimitris Theodoropoulos, Catalin Bogdan Ciobanu, Georgi Kuzmanov

6. S. Vassiliadis, S. Wong, G.N. Gaydadjiev, K.L.M. Bertels, G.K. Kuzmanov, and E. Moscu Panainte, The molen polymorphic processor

7. G.K. Kuzmanov, The molen polymorphic media processor, Ph.D. thesis, Delft University of Technology, December 2004

8. DWARV: DELFTWORKBENCH AUTOMATED RECONFIGURABLE VHDL GENERATOR, Yana Yankova, Georgi Kuzmanov, Koen Bertels, Georgi Gaydadjiev, Yi Lu, Stamatis Vassiliadis

9. Automated HDL Generation: Comparative Evaluation, Yana Yankova, Koen Bertels, Stamatis Vassiliadis, Roel Meeuws, Arcilio Virginia

10. DWARV version 1.2, C Language Restrictions , Yana Yankova, CE Laboratory, TU Delft

11. DWARV Tool Set User Manual, version 1.2,Yana Yankova, CE Laboratory, TU Delft

12. J. W. Cooley and J. W. Tukey, An Algorithm for the Machine Calculation of Complex Fourier Series, Math. Comp., vol. 19, pp.297-301, April 1965.

13. A. V. Oppenheim and R. W. Schafer, Discrite-Time Signal Processing, Prentice-Hall Inc., 1999.

14. V. Sorensen, M. T. Heideman and C. S. Burrus, On Computing the Split-radix FFT, IEEE Trans. Acoust., Speech, Signal Processing, vol. ASSP-34, pp. 152-156, Feb. 1986.

15. P. Duhamel, Algorithms Meeting the Lower Bound on the Multiplicative Complexity of Length-2n DFTs and Their Connection with Paractical Algorithms, Electron Letters, vol. 38,

No. 9, pp. 1504-1511, Sep. 1990.

16. S. He and M. Tokelson, A New Approach to Pipeline FFT Processor, Parallel Processing Symposium, The 10th International, pp. 766-770, April 1996.

17. E. H. Wold and A. M. Despain, Pipeline and Parallel-Pipeline FFT Processors for VLSI Implementation, IEEE Transactions on Computers, vol. 33, No.5, pp.414-426, May 1984.

18. Y. Ma, An Effective Memory Addressing Scheme for FFT Processor, IEEE Transaction on Signal Processing, vol. 47, Issue: 3, pp. 907-911, Mar. 1999.

19. C. L. Wang and C. H. Chang, A New Memory-Based FFT Processor for VDSL Transceivers, IEEE International Symposium on Circuits and System, vol. 4, pp. 670-673, 2001.

20. S. He and M. Tokelson, Design and Implementation of 1024-point FFT Processor, Proc. IEEE Custom Integrated Circuit Conference, pp. 131-134

21. G.. Bi and E. V. Jones, A Pipelined FFT Processor for Word Sequential Data, IEEE Transaction on Acoustics, Speech, Signal Processing, vol. 37, No. 12, pp. 1982-1985, Dec. 1989.

22. C. P. Hung, S. G. Chen and K. L. Chen, Design of An Efficient Variable-Length FFT Processor, Circuit System, vol. 2 pp. -833-836, May 2004.

23. H. F. Lo, M. D. Shieh, and C. M. Wu, Design of an Efficient FFT Processor for DAB System, IEEE International Symposium on Circuits and System, vol. 4, pp. 654-654, 2001.

24. B. S. Kim and L. S. Kim, Low Power Pipelined FFT Architecture for Synthetic Aperture Radar Signal Processing, in Proc. IEEE Midwest Symposium on Circuit and Systems, vol.3, pp. 136-1370, 1996.

25. A. Delaruelle, J. Huisken, J. V. Loon, F. Welten, A Channel Demodulator IC for Digital Audio Broadcasting, Proc. IEEE Custom Integrated Circuits Conference, pp. 47-50, 1994.

26. C. H. Chang, C. L. Wang and Y. T. Chang, Efficient VLSI Architectures for Fast Computation of the Discrete Fourier Transform and its Inverse, IEEE Transaction on Signal Processing, vol. 48, Issue: 11, pp. 3206-3216, Nov. 2000.

27. D. Cohen, Simplified Control of FFT Hardware, IEEE Trans. Acoust., Speech Signal Processing, vol. ASSP-24, pp. 577-579, Dec. 1976

28. Y. Ma and W. Lars, A Hardware Efficient Control of Memory Addressing for High-Performance FFT Processor, IEEE Transaction on Signal Processing, vol. 48, Issue: 3, pp. 917-921, Mar. 2000.

29. J. A. Hiadalgo, J. Lopez, F. Aruguello, and E. L. Zapata, Area-efficient Architecture for Fast Fourier Transform, IEEE Transaction. Circuit System.- ., vol. 46 No. 2, pp. 187-193, Feb. 1999.

30. B. M. Baas, An approach to low power, high performance, fast Fourier transform processor design, [PhD Thesis], Stanford University, 1999.

31. FFT MegaCore function user guide, Version 1.02: Altera Inc.,March 2001.

32. High-performance 1024-point complex FFT/IFFT, Version 2.0, Xilinx Inc., July 2000.

33. B. M. Baas, FFT Processor Chip Info Page, available at http://www-star.stanford.edu/ bbaas/fftinfo.html, 2004.

34. Reconfigurable Computing: What, Why, and Implications for Design Automation Andre DeHon and John Wawrzynek

35. V. Pulkki, Compensating displacement of amplitude-panned virtual sources, in Proc. of the AES 22nd Int. Conference, Audio Engineering Society, 2002, pp. 186195.

36. M. A. Gerzon, Ambisonics in multichannel broadcasting and video, J. Acoust. Soc. Am.,vol. 33, pp. 859871, Nov. 1985.

37.E. N. G. Verheijen, Sound Reproduction by Wave Field Synthesis. PhD thesis, Delft University of Technology, 1997.