DESIGN AND IMPLEMENTATION OF AN ASYNCHRONOUS PIPELINED FFT PROCESSOR

Master's thesis project at

Electronics Systems

Jonas Claeson

Reg nr: LiTH-ISY-EX-3356-2003

Linköping, June 13, 2003

DESIGN AND IMPLEMENTATION OF AN ASYNCHRONOUS PIPELINED FFT PROCESSOR

Examensarbete utfört i Elektroniksystem
vid Linköpings Tekniska Högskola
av

Jonas Claeson

Reg nr: LiTH-ISY-EX-3356-2003

Supervisor: Weidong Li
Examiner: prof. Lars Wanhammar
Linköping, June 12, 2003

Avdelning, Institution
Division, department

Department of Electrical Engineering
581 83 LINKÖPING

Datum
Date

2003-06-06

Språk
Language

☐ Svenska/Swedish
☒ Engelska/English
☐ _____

Rapporttyp
Report: category

☐ Licentiatavhandling
☒ Examensarbete
☐ C-uppsats
☐ D-uppsats
☐ Övrig rapport
☐ _____

URL för elektronisk version

http://www.ep.liu.se/exjobb/isy/2003/3356

ISBN

ISRN LiTH-ISY-EX-3356-2003

Serietitel och serienummer
Title of series, numbering

ISSN

Titel
Title

Design och implementering av en asynkron pipelinad FFT processor

Design and Implementation of an Asynchronous Pipelined FFT Processor

Författare
Author

Jonas Claeson

Sammanfattning
Abstract

FFT processors are today one of the most important blocks in communication equipment. They are used in everything from broadband to 3G and digital TV to Radio LANs. This master's thesis project will deal with pipelined hardware solutions for FFT processors with long FFT transforms, 1k to 8k points. These processors could be used for instance in OFDM communication systems.

The final implementation of the FFT processor uses a GALS (*Globally Asynchronous Locally Synchronous*) architecture, that implements the SDF (*Single Delay Feedback*) radix-2^2 algorithm.

The goal of this report is to outline the knowledge gained during the master's thesis project, to describe a design methodology and to document the different building blocks needed in these kinds of systems.

Nyckelord
Keywords

DFT, FFT, Pipelined, Parameterizable, Processor, GALS, Radix-2^2, SDF

Abstract

FFT processors are today one of the most important blocks in communication equipment. They are used in everything from broadband to 3G and digital TV to Radio LANs. This master's thesis project will deal with pipelined hardware solutions for FFT processors with long FFT transforms, 1k to 8k points. These processors could be used for instance in OFDM communication systems.

The final implementation of the processor uses a GALS (Globally Asynchronous Locally Synchronous) architecture, that implements the SDF (*Single Delay Feedback*) radix-2^2 algorithm.

The goal of this report is to outline the knowledge gained during the master's thesis project, to describe a design methodology and to document the different building blocks needed in these kinds of systems.

Acknowledgements

First of all I would like to thank my examiner professor Lars Wanhammar for giving me this interesting master's thesis project and for the general directions of my work. I would also like to thank my supervisor Weidong Li for his help with more detailed questions. Two other persons that helped me a lot is Kent Palmkvist with VHDL and synthesis related questions, and Jonas Carlsson with questions concerning asynchronous circuits.

Terminology

Table: Terminology.

Abbreviation or term	Explanation
BFP	*Block Floating Point*. One way of representing data internally.
butterfly	Basic building block in HW FFT processors.
CG FFT	*Constant Geometry* FFT.
COFDM	*Coded Orthogonal Frequency Division Multiplexing*.
DFT	*Discrete Fourier Transform*. The discrete version of the continuous fourier transform. Transforms a signal from a time-domain to a frequency-domain.
DFT	*Design For Test*. Extra HW is added in the design to ease and speed up the testing.
DIF	*Decimation In Frequency.* One out of two ways of implementing a radix PE.
DIT	*Decimation In Time*. One out of two ways of implementing a radix PE.
FFT	*Fast Fourier Transform*. Quick way of computing a DFT.
GALS	*Globally Asynchronous Locally Synchronous.* A way of decomposing a system into several synchronous blocks that communicate with an asynchronous protocol.
HW	*Hardware*.
in-place algorithm	Output of a butterfly is written back to where the input came from.
LS-system	*Locally Synchronous* system.
MDC	*Multipath Delay Commutator.* Block between radix PEs in a pipelined architecture.

Table: Terminology.

Abbreviation or term	Explanation
not-in-place algorithm	Output of a butterfly is not written back to where the input came from.
OFDM	*Orthogonal Frequency Division Multiplexing*. OFDM is a broadband multicarrier modulation method used in a lot of communication systems.
PE	*Processing Element*.
SDC	*Single-path Delay Commutator*. Block between radix PEs in a pipelined architecture.
SDF	*Single-path Delay Feedback*. Block between radix PEs in a pipelined architecture.
SFG	*Signal Flow Graph*. Describes an algorithm in a graphical way using adders, multipliers, signal wires, etc.
SIC	*Single Instruction Computer*.
SNR	*Signal to Noise Ratio*. Not a good measurement in the FFT context.

Notation

Table: Symbols.

Symbol	Explanation
N	Length of the input and output sequence of a DFT or FFT.
x	Input signal to an FFT processor.
X	FFT transform of the input signal x.

Table: Operators and functions.

Operator or function	Explanation
a|b	b is divisible by a, i.e. b/a gives 0 in rest.
$<X>_N$	X modulo N.

Table of Contents

1 Introduction

1.1 General

FFT processors are involved in a wide range of applications today. Not only as a very important block in broadband systems, digital TV, etc., but also in areas like radar, medical electronics and the SETI project (Search for Extraterrestrial Intelligence).

Many of these systems are real-time systems, which means that the systems has to produce a result within a specified time. The work load for FFT computations are also high and a better approach than a general purpose processor is required, to fulfill the requirements at a reasonable cost. For instance using application specific processors, algorithm specific processors, or ASICs could be the solution to these problems. In this master's thesis project an ASIC FFT processor will be designed. ASIC is the choice because of its lower power consumption and higher throughput.

1.2 Scope of the Report

The report is concentrated on pipelined FFT processors, and what architectures and algorithms that are most suitable for dedicated FFT processors.

The first part of the report gives a review on the theory behind the DFT and FFT algorithm and different approaches to implement the FFT in HW. Some terminologies like radix butterflies, pipelining, commutators, algorithms, architectures, etc., are introduced in this part.

The second part of the report describes the main goal of this master's thesis project, i.e. to design and implement a parameterized pipelined FFT processor for transform lengths from 1k to 8k samples per frame. These transform lengths and the parameterization reduces the amount of algorithms, architectures, and so on, that could be taken into account when designing a processor according to these criteria. Some parts of the theory are therefor very briefly described compared to others, because of its limited usefulness in the considered area.

What trade-offs have to be made? What architecture and algorithm should be used? What types of simulations should be done? How is testing performed? These are some of the questions that will be discussed in the second part.

1.3 Project Requirements

The requirements for this master's thesis project are as follows:

1. The transform length shall be able to vary between 1k and 8k samples in powers-of-2.
2. The input signal shall be a continuous data stream.
3. The input signal shall consist of only one continuous data stream.
4. The word length of the input and output signal shall be parameterizable. The internal word lengths shall also be parameterizable.
5. Safe scaling shall be used.
6. Data shall be represented with two's complement format.
7. The implemented architecture shall be pipelined.

These are the prioritizations that should be taken mostly into account:

- Effect, power consumption and throughput are superior to die area and latency within reasonable limits. Latency is hard to affect when the input stream arrives continuously.
- SNR is a poor quality measurement in FFT processors, this measurement should therefor not be considered too important in the design. Though, this does not mean that the output can have too low precision due to quantization noise.

These are the first requirements on the FFT processor that is going to be designed. Later in the report new restrictions and requirements will be added to narrow down the area of investigation even further, to concentrate the work on the type of architecture found to be most adequate.

1.4 Reading Instructions

This list gives a short description of the content of each chapter.

- *Chapter 1 Introduction* contains the introduction of the project. What will the project be all about? What will the result of the project be?
- *Chapter 2 Algorithms* contains a description of a lot of different FFT algorithms, not only those that will be considered in the project, but also a few other ones.
- *Chapter 3 Architectures* contains a description of a lot of different FFT architectures that implement the FFT algorithms. Also here some algorithms that not will be considered in the project will be described, along with all the more adequate ones.
- *Chapter 4 Numerical Effects* contains a basic theoretical introduction on the quantization errors in FFT algorithms and architectures.
- *Chapter 5 Implementation Choices* contains an explanation why the radix-2^2 algorithm and the SDF architecture is chosen to be to one being implemented.
- *Chapter* 6 Radix-2^2 FFTs contains the derivation of the radix-2^2 algorithm and architectural descriptions of its components.
- *Chapter 7 FFT Design* contains the design methodology used in this project. Some problems that arose during the project and the solution are discussed in this chapter.
- *Chapter 8 Asynchronous Design* contains a very basic introduction to asynchronous circuits, with a focus on GALS. The methodology is the focus in this chapter, but it also describes the asynchronous architecture of the final implementation of the asynchronous FFT processor.
- *Chapter 9 Future Work* contains suggestions of what the next steps in this project could be.

- *Chapter 10 Summary* contains the summary for the whole project. General thoughts and acquired knowledge are discussed.
- *Chapter 11 Bibliography* contains the references referred to, inside square brackets, in the text.

2 Algorithms

2.1 Introduction

The algorithms chapter will introduce the DFT definition, the FFT algorithm and different approaches to compute FFTs in HW. The discussion will mainly be focused on FFT algorithms useful for long FFTs, but other algorithms, will also be described briefly.

2.2 The DFT Algorithm

A DFT is a transform that is defined as

$$X(k) = \sum_{n=0}^{N-1} x(n) \cdot W_N^{nk}, k \in [0, N-1] \quad \text{(Eq 2.1)}$$

where

$$W_N = e^{-j2\frac{\pi}{N}} \quad \text{(Eq 2.2)}$$

is the *N*-th root of unity. The inverse of the DFT (IDFT) is defined as

$$x(n) = \frac{1}{N} \sum_{k=0}^{N-1} X(k) \cdot W_N^{-nk}, n \in [0, N-1] \quad \text{(Eq 2.3)}$$

These equations show that the complexity of a direct computation of DFTs and IDFTs is O(N^2), hence the long transforms considered in this master's thesis will be very costly in a straight forward computation. The

FFT algorithm deals with these complexity problems by exploiting regularities in the DFT algorithm.

2.3 FFT Algorithms

An FFT algorithm uses a divide-and-conquer approach to reduce the computation complexity for DFT, i.e. one big problem is divided into a lot of different smaller problems that in the end are assembled to the solution of the original problem.

In a communication system that uses an FFT algorithm there is also a need for an IFFT algorithm. Since the DFT and the IDFT are similar both of these can be computed using basically the same FFT HW, swap the real and imaginary parts of the input, compute the FFT and swap the real and imaginary data of the output. The output is now the IFFT of the input data, except for the scaling factor in the IFFT algorithm, 1/N. Usually this is not a problem, and this will therefor not be discussed henceforth.

2.4 Common Factor Algorithms

Common factor algorithms are one way of dividing the problem, using the divide-and-conquer approach. This method is the most widely used way of computing FFTs. *N* is then divided into factors according to:

$$N = \prod_i N_i \qquad \text{(Eq 2.4)}$$

where the factors are constrained in the following way:

$$\exists a \forall i (a | N_i) \qquad \text{(Eq 2.5)}$$

This basically means that they have one factor in common. In this way an FFT can be computed in *i* number of steps. There are two equally computational complex algorithms that can be derived from this, DIF (decimation-in-frequency) and DIT (decimation-in-time).

2.5 Radix-2 Algorithm

The radix-2 algorithm is a special case of the common factor algorithm for *N*-point DFTs, where *N* is power-of-2. To derive the radix-2 algorithm, the indices *n* and *k* in Equation 2.1 are represented by

$$n = 2^{\alpha-1} \cdot n_{\alpha-1} + 2^{\alpha-2} \cdot n_{\alpha-2} + \ldots + n_0 = \sum_{\beta=0}^{\alpha-1} 2^{\beta} \cdot n_{\beta} \qquad \text{(Eq 2.6)}$$

$$k = 2^{\alpha-1} \cdot k_{\alpha-1} + 2^{\alpha-2} \cdot k_{\alpha-2} + \ldots + k_0 = \sum_{\beta=0}^{\alpha-1} 2^{\beta} \cdot k_{\beta}$$

$$n_i, k_i \in \{0, 1\}, i = 0 \ldots \alpha - 1$$

where

$$N = 2^{\alpha} \qquad \alpha \in \aleph \qquad \text{(Eq 2.7)}$$

When these representations are used for substitution in Equation 2.1, the DFT definition can be rewritten as

(Eq 2.8)

$$\cdot(k_{\alpha-1}, k_{\alpha-2}, \ldots, k_0) = \sum_{n_0=0}^{1} \sum_{n_1=0}^{1} \cdots \sum_{n_{\alpha-1}=0}^{1} x(n_{\alpha-1}, n_{\alpha-2}, \ldots, n_0) \cdot W_N^{\left(\sum_{\beta=0}^{\alpha-1} 2^{\beta} \cdot n_{\beta}\right) \cdot \left(\sum_{\beta=0}^{\alpha-1} 2^{\beta} \cdot k_{\beta}\right)}$$

The last term in the right side of Equation 2.8 can be expressed as

(Eq 2.9)

$$W_N^{\left(\sum_{\beta=0}^{\alpha-1} 2^{\beta} \cdot n_{\beta}\right) \cdot \left(\sum_{\beta=0}^{\alpha-1} 2^{\beta} \cdot k_{\beta}\right)} = W_N^{2^{\alpha-1} k_{\alpha-1} \cdot \sum_{\beta=0}^{\alpha-1} 2^{\beta} \cdot n_{\beta}} \cdot W_N^{2^{\alpha-2} k_{\alpha-2} \cdot \sum_{\beta=0}^{\alpha-1} 2^{\beta} \cdot n_{\beta}} \cdot \ldots \cdot W_N^{k_0 \cdot \sum_{\beta=0}^{\alpha-1} 2^{\beta} \cdot n_{\beta}}$$

Observe that

$$W_N^N = \left(e^{-\frac{j2\pi}{N}}\right)^N = 1 \qquad \text{(Eq 2.10)}$$

By using Equation 2.10 on the different factors of Equation 2.9 the following relations are found

$$G_0 = W_N^{2^{\alpha-1}k_{\alpha-1}\cdot\sum_{\beta=0}^{\alpha-1}2^\beta\cdot n_\beta} = W_N^{2^{\alpha-1}k_{\alpha-1}\cdot\sum_{\beta=0}^{0}2^\beta\cdot n_\beta} = W_N^{2^{\alpha-1}k_{\alpha-1}n_0} \quad \text{(Eq 2.11)}$$

$$G_1 = W_N^{2^{\alpha-2}k_{\alpha-2}\cdot\sum_{\beta=0}^{\alpha-1}2^\beta\cdot n_\beta} = W_N^{2^{\alpha-2}k_{\alpha-2}\cdot\sum_{\beta=0}^{1}2^\beta\cdot n_\beta}$$

$$\dots$$

$$G_{\alpha-1} = W_N^{k_0\cdot\sum_{\beta=0}^{\alpha-1}2^\beta\cdot n_\beta}$$

Insert Equation 2.11 in Equation 2.8

$$X(k_{\alpha-1}, k_{\alpha-2}, \dots, k_0) = \sum_{n_0=0}^{1}\sum_{n_1=0}^{1}\cdots\sum_{n_{\alpha-1}=0}^{1} x(n_{\alpha-1}, n_{\alpha-2}, \dots, n_0)\cdot\prod_{i=0}^{\alpha-1} G_i \quad \text{(Eq 2.12)}$$

This summation can be divided into sequential summations

$$x_1(k_0, n_{\alpha-2}, n_{\alpha-3}, \dots, n_0) = \sum_{n_{\alpha-1}=0}^{1} x(n_{\alpha-1}, n_{\alpha-2}, \dots, n_0)\cdot G_{\alpha-1} \quad \text{(Eq 2.13)}$$

$$x_2(k_0, k_1, n_{\alpha-3}, \dots, n_0) = \sum_{n_{\alpha-2}=0}^{1} x_1(k_0, n_{\alpha-2}, n_{\alpha-3}, \dots, n_0)\cdot G_{\alpha-2}$$

$$\dots$$

$$x_{\alpha-1}(k_0, k_1, \dots, k_{\alpha-1}) = \sum_{n_0=0}^{1} x_{\alpha-2}(k_0, \dots, k_{\alpha-2}, n_0)\cdot G_0$$

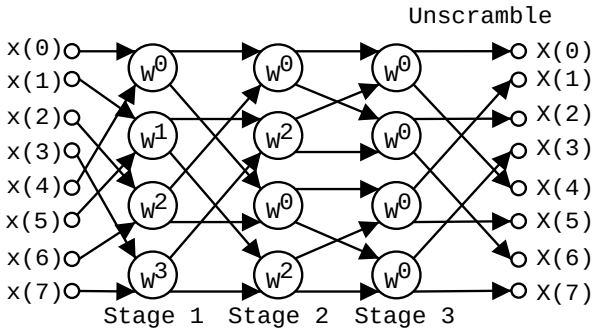
Finally, to obtain the FFT an unscrambling stage is added to reorder the output data in natural order. Unscrambling is done by bit-reversing.

$$X(k_{\alpha-1}, k_{\alpha-2}, \dots, k_0) = x_{\alpha-1}(k_0, k_1, \dots, k_{\alpha-1}) \quad \text{(Eq 2.14)}$$

With this algorithm the computational complexity is reduced to $O(N\log_2(N))$ butterfly operations. The computation has also been divided into $\log_2(N)$ different steps, which is an advantage considering pipelining in HW.

The SFG for this derivation of the FFT algorithm looks like Figure 2.1 for an 8-point radix-2 DIF FFT:

Figure 2.1: SFG for an 8-point radix-2 DIF FFT.



2.6 Radix-*r* Algorithm

The radix-*r* algorithm uses the same approach as radix-2, but with the decomposition using base-*r* instead of base-2. *N* is factorized as

$$N = r^{\alpha} \qquad r, \alpha \in \aleph \qquad \text{(Eq 2.15)}$$

The derivation of the radix-*r* is analogous to the derivation of radix-2. The proof will therefore be left out here. The computational complexity for the radix-*r* case is $O(N\log_r(N))$ butterfly operations divided into $O(\log_r(N))$ butterfly stages.

2.7 Split Radix Algorithm

The split radix algorithm is one way of decreasing the number of multiplications and additions required, [1]. The main drawback is the more irregular structure compared to mixed radix and constant radix algorithms. Because of the irregular structure this algorithm is not suitable for parameterization, and will therefore not be studied more thoroughly.

2.8 Mixed Radix Algorithm

Mixed radix algorithms is a combination of different radix-r algorithms. That is, different stages in the FFT computation have different radices. For instance, a 16-point long FFT can be computed in two stages using one stage with radix-8 PEs, followed by a stage of radix-2 PEs. This adds

a bit of complexity to the algorithm compared to radix-r, but in return it gives more options in choosing the transform length.

2.9 Prime Factor Algorithms

Prime factor algorithms decompose N into factors that are relative prime, which means that the greatest common divisor of the factors is equal to 1. There are two reasons why prime factor algorithms will not be considered later in the report. Firstly, it restricts the transform length in a way that N cannot be power-of-2, which is a requirement. Secondly, it doesn't scale very good, because for large Ns the decomposing relative prime numbers will also be large, hence will result in a very complex implementation of the PEs.

2.10 Radix-*r* Butterflies

The radix-*r* butterflies are the blocks that perform the basic computations in the radix-*r* algorithm. The following reasoning will explain how the SFG structure is derived (for the radix-2 case). Only the proof of the first stage will be shown, the other proofs are analogous. The butterfly obtained is a radix-2 DIF (decimation-in-frequency). From Equation 2.11 and Equation 2.13

$$x_1(k_0, n_{\alpha-2}, \ldots, n_0) = \sum_{n_{\alpha-1}=0}^{1} x(n_{\alpha-1}, n_{\alpha-2}, \ldots, n_0) \cdot W_N^{k_0 \cdot \sum_{\beta=0}^{\alpha-1} 2^\beta \cdot n_\beta} \qquad \text{(Eq 2.16)}$$

$$= \sum_{n_{\alpha-1}=0}^{1} x(n_{\alpha-1}, n_{\alpha-2}, \ldots, n_0) \cdot W_N^{k_0 \cdot 2^{\alpha-1} \cdot n_{\alpha-1}} \cdot W_N^{k_0 \cdot \sum_{\beta=0}^{\alpha-2} 2^\beta \cdot n_\beta}$$

The last factor in the summation is not depending on the summation variable, hence this factor can be lifted out from the summation

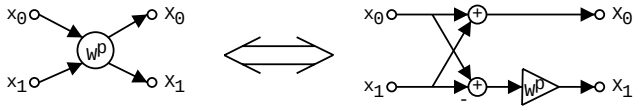
$$= W_N^{k_0 \cdot \sum_{\beta=0}^{\alpha-2} 2^{\beta} \cdot n_{\beta}} \cdot \sum_{n_{\alpha-1}=0}^{1} x(n_{\alpha-1}, n_{\alpha-2}, \ldots, n_0) \cdot W_N^{k_0 \cdot 2^{\alpha-1} \cdot n_{\alpha-1}} \qquad \text{(Eq 2.17)}$$

$$= \left\{ W_N^{k_0 \cdot 2^{\alpha-1} \cdot n_{\alpha-1}} = e^{\frac{-j2\pi}{2^{\alpha}} \cdot \frac{2^{\alpha}}{2} \cdot k_0 \cdot n_{\alpha-1}} = (-1)^{k_0 \cdot n_{\alpha-1}} \right\}$$

$$= W_N^P \cdot (x(0, n_{\alpha-2}, \ldots, n_0) + (-1)^{k_0} \cdot x(1, n_{\alpha-2}, \ldots, n_0))$$

According to the above computations, the basic FFT computations can be made with a structure called radix element. Radix elements for higher radixes can be derived in a similar way. These elements will have *r* inputs and *r* outputs for a radix-r element. The figure below shows the structure for the radix-2 case.

Figure 2.2: Structure of a radix-2 DIF butterfly.



3 Architectures

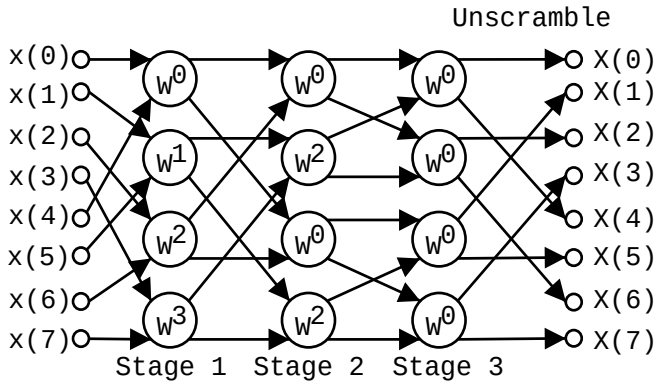
3.1 Introduction

This chapter discusses different architectures used for FFT computations. As in chapter 2 Algorithms, mostly the architectures useful for long FFTs will be taken into account. Their advantages and drawbacks will be discussed.

3.2 Array Architectures

The array architecture can only be used for very short FFTs, because of the extensive use of chip-area. This comes from the use of one processing element (PE) for each butterfly in the signal flow graph (SFG). Normally FFTs longer than 16 points are not implemented with this architecture, hence it will not be discussed in details.

Figure 3.1: SFG for an 8-point array FFT architecture.

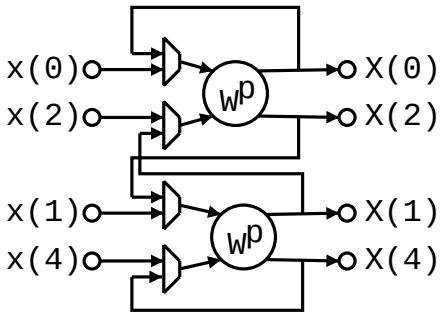


3.3 Column Architectures

The column architecture uses an approach that requires less area on the chip than the array architecture. It is done by collapsing all the columns in an array architecture into one column, hence a new frame cannot be processed until the processing of the current frame is finished. Hence, this architecture is not suitable for pipelining. The area requirement is obviously smaller, only N/r radix-r elements, than for the array architecture. The architecture is still not small enough to be taken into account for long FFTs.

An architectural structure of a 4-point radix-2 DIF FFT can be seen below. To get a simple feedback network a type of structure called constant geometry FFT (CG FFT) is often used as a starting point. This means that the connection network in an array architecture would be the same between all stages.

Figure 3.2: Structure of a 4-point radix-2 column architecture.



3.4 Pipelined Architectures

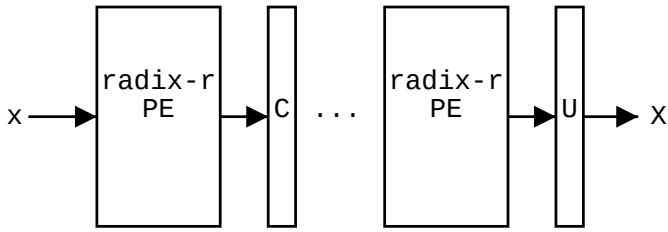
Pipelined architectures are useful for FFTs that require high data throughput. The basic principle with pipelined architectures is to collapse the rows, instead of the stages like in column architectures. The architecture is built up from radix butterfly elements with commutators in between. An unscrambling stage is sometimes added on the input or output side of the processor, if the output data is needed to be in natural order.

The advantage with these architectures, are for instance, high data throughput, relatively small area and a relatively simple control unit.

These advantages make this solution suitable for the long FFTs considered in this master's thesis project.

The basic structure of the pipelined architecture is shown below. Between each stage of radix-r PEs there is a commutator (denoted C in the picture). The last stage is the unscrambling stage (denoted U in the picture). The commutator reorders the output data from the previous stage and feeds to the following stage. The unscrambler rearranges the data in natural sorted order.

Figure 3.3: General structure of a pipelined FFT architecture.



3.4.1 MDC, SDF and SDC Commutators

There are basically three kinds of commutators, Multipath Delay Commutator (MDC), Single-path Delay Feedback (SDF) and Single-path Delay Commutator (SDC). They all give the architecture different properties, especially when it comes to total memory requirement.

A commutator is a switch for data between the radix butterfly stages in the pipeline. It stores parts of the FFT computations temporarily in order to perform the switching properly. The SDF commutator is somewhat different, because it also feeds data backwards, Figure 3.5.

The figures below show the structure of the commutators. In these figures 'a' denotes the stage number in the pipeline. The numbers in the boxes gives the size of that FIFO buffer in complex samples. C2 is a switch and BF4 is short for radix-4 butterfly element.

Figure 3.4: Multipath Delay Commutator structure.

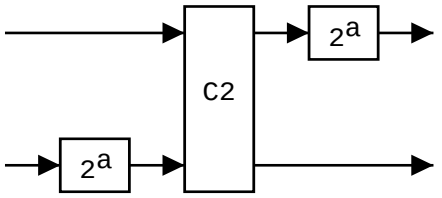


Figure 3.5: Single-path Delay Feedback Commutator structure.

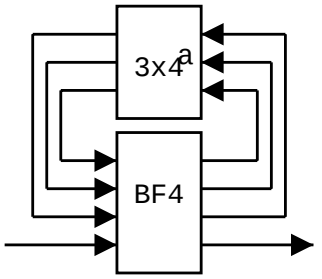
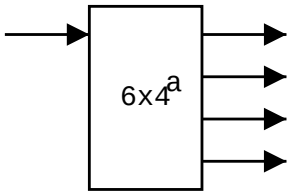


Figure 3.6: Single-path Delay Commutator structure.



3.4.2 Pipeline Architecture Comparisons

There are many different pipelined architectures. They have different memory requirements, different complexities, different utilization, etc. A summary of the most common pipelined architectures are show in Table 3.1, [2]. The abbreviations of the architecture names are composed

in the following way, e.g. R2MDC is short for radix-2 multipath delay commutator FFT architecture.

Table 3.1: Pipeline architecture comparison.

Architecture	Multiplier #	Adder #	Memory size	Control
R2MDC	$2(\log_4 (N-1))$	$4\log_4 N$	3N/2 - 2	Simple
R2SDF	$2(\log_4 (N-1))$	$4\log_4 N$	N - 1	Simple
R4SDF	$\log_4 (N-1)$	$8\log_4 N$	N - 1	Medium
R4MDC	$3(\log_4 (N-1))$	$8\log_4 N$	5N/2 - 4	Simple
R4SDC	$\log_4 (N-1)$	$3\log_4 N$	2N - 2	Complex
$R2^2SDF$	$\log_4 (N-1)$	$4\log_4 N$	N - 1	Simple

The $R2^2SDF$ architecture is interesting. When it comes to these properties in the table this architecture is equal to or better than the other architectures, with one exception, the number of adders is 25% lower in the R4SDC architecture. The $R2^2SDF$ architecture will therefore be a really good candidate to investigate, when choosing the architecture that will be implemented.

3.5 Multipipelined Architectures

Multipipelined architectures are built up in a similar way as normal pipelined architectures, but with the distinction that some stages in the pipeline can use two or more radix butterfly elements.

These architectures achieve a higher parallelism than regular pipelined architectures, [5]. The improvement in parallelism is equal to the number of pipes introduced.

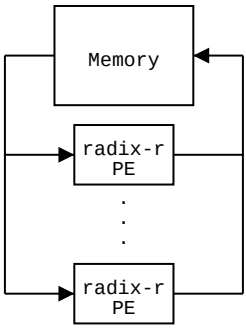
3.6 SIC FFT Architectures

SIC FFT Architectures can be a good choice when throughput requirements are not high compared with the throughput of available butterflies, [1]. In this architecture all the butterfly elements share the same memory. A radix PE reads data from the memory and when it is finished with the computation it writes the data back to the memory. This results in a lot of

memory accesses, which could be both hard to implement and be costly in power consumption.

The architecture can be adapted to the requirements specification more precisely by adapting the number of radix PEs. For some specifications this architecture reduces radix PEs, which reduces both the die area and the power consumption.

Figure 3.7: Structure of the SIC FFT architecture.



3.7 Cached-FFT Architectures

Cached-FFT architectures are mainly used for reducing the power consumption, [4]. The idea is to use a cache-memory between the radix-r PEs and the main memory to decrease the number of main memory accesses, which is very energy consuming.

4 Numerical Effects

4.1 Introduction

DSP systems almost always suffer from quantization effects, because of the limited internal data word length. For instance, a multiplication by two operands always gives a result that is longer in bits than each of the operands. The result have to be truncated or rounded to avoid long internal word length, hence quantization occurs. Quantization is explained in Section 4.3 on page 21.

There is another thing in FFT systems that have to be considered, and it is overflow. If an overflow occurs in an FFT system it will generate a faulty output. How this is solved will be discussed in Section 4.2 on page 19.

4.2 Safe Scaling

To prevent operations to overflow and cause errors a method called safe scaling is used. It means that the output from each radix PE and the input to the first FFT stage are scaled in such a way, that for certain the computation in the next radix PE will not overflow. The scaling is often a division by a power-of-2 number, because it easily can be implemented by arithmetic right shift.

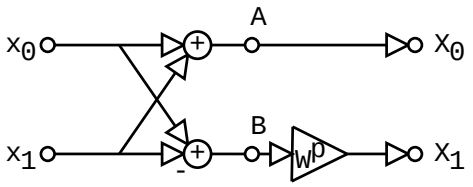
To simplify the discussion about safe scaling, the radix-2 case will be used as an example. The safe scaling for the radix-*r* FFT algorithm is given without detailed discussion.

4.2.1 Radix-2 Safe Scaling

In a radix-2 butterfly element, overflow can occur for signals in wire A and B, see Figure 4.1, after the summations. Overflow can, in reality, not occur after the twiddle factor multiplication, because the absolute value of the twiddle factor is always very close to one. Hence, the absolute value does not change, only the argument.

Fractional two's complement is going to be used in this FFT processor, therefore signals that can be represented is in the range of [-1, 1[. The absolute value after each summation can in the worst case be twice as big as the input. Hence, to prevent the output from overflow the absolute value of the input signals have to be smaller than 0.5. One way to ensure that the input signals are in this range is to divide the output signals in the previous butterfly stage by a factor of 2. This method will always prevent overflow and it only requires a little extra HW, hence, this method is used in the FFT implementation.

Figure 4.1: Problem areas in a radix-2 element.



No overflow will now occur in the radix-2 butterflies using this method, except for the first butterfly element. The input to this butterfly also has to be scaled. The real and imaginary input to the FFT processor will be in the range [-1, 1[, the absolute value could therefore be as large as $2^{0.5}$. In principle the input should be scaled by a factor of $1/2^{1.5}$, to get the input value of the first radix PE in the range [-0.5, 0.5[. This division is not cheap to implement in HW and a scaling factor of 1/4 is a better choice, because it can be implemented using arithmetic right shift.

The absolute value of the output of the FFT processor is smaller than 0.5, due to the last safe scaling. To use the whole range of representable values a last stage called final scaling is often added. This stage performs a multiplication by 2, increasing the absolute value of the output to the range [-1, 1[.

4.2.2 Radix-*r* Safe Scaling

Radix-*r* safe scaling is similar to radix-2 safe scaling. The only difference is that the scaling factor in the radix elements are 1/*r* instead of 1/2, the prescaling stage is 1/2*r* instead of 1/4 and the final scaling is a multiplication of *r* instead of 2.

4.3 Quantization

Quantization occurs after each multiplication in the radix PEs. The errors introduced by quantization are modelled with a technique called noise modelling. In this technique stochastic noise sources are added to the SFG where quantization occurs. From the new SFG statistical calculations can be made to estimate the amount of noise introduced in the output by quantization.

4.3.1 Two's Complement Quantization

The quantization can be done either by rounding or by truncation. These approaches give different statistical properties on the quantization error. The representation of a fractional two's complement number is given by

$$\bar{x} = -x_0 + \sum_{i=1}^{W_d - 1} x_i \cdot 2^{-i} \qquad x_i \in \{0, 1\} \qquad \text{(Eq 4.1)}$$

This definition shows that the values it can represent is uniformly distributed in the [-1, 1[interval, hence the quantization error does not depend on the magnitude of the value. The difference between two adjacent values is equal to the truncation error. Hence, the truncation error has a non-zero expectation value.

$$0 \le \Delta_t \le \frac{1}{2^{W_d - 1}} \qquad \text{(Eq 4.2)}$$

Rounding is a better quantization method, because the expectation value of the rounding error is zero.

$$-\frac{1}{2^{W_d}} \le \Delta_r \le \frac{1}{2^{W_d}} \qquad \text{(Eq 4.3)}$$

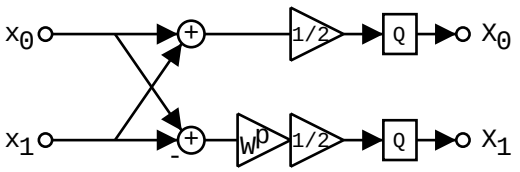
Assuming that the data is uniformly distributed in the range of [-1,1[, the errors are evenly distributed in the above given intervals. The variance of a stochastic variable like this is

$$\sigma^2 = \frac{1}{12} \cdot \left(\frac{1}{2^{W_d}} \right)^2 \qquad \text{(Eq 4.4)}$$

4.3.2 Radix-2 Quantization

The quantization discussion in this section is only considering DIF radix butterfly elements with rounding and safe scaling. The scaling gives two properties, the first is that no quantization occurs in the adders and the second that quantization occurs at both output nodes. The adders never cause overflow with safe scaling and both output nodes have multiplications when using safe scaling.

Figure 4.2: Quantization in a radix-2 DIF PE.



The quantization, denoted Q, in Figure 4.2 can be modelled as an adder adding a complex stochastic variable *n* to the original signal. The real part and the imaginary part can be seen as two independent stochastic variables.

$$n = n_{re} + j \cdot n_{im} \qquad \text{(Eq 4.5)}$$

$$E\{n_{re}\} = E\{n_{im}\} = 0$$

$$V\{n_{re}\} = V\{n_{im}\} = \frac{1}{12} \cdot \left(\frac{1}{2^{W_d}} \right)^2$$

The expectation value and variance of the complex noise are

$$E\{n\} = E\{n_{re} + j \cdot n_{im}\} = 0 \qquad \text{(Eq 4.6)}$$

$$V\{n\} = E\{n \cdot \bar{n}\} = E\{n_{re}^2 + n_{im}^2\} = E\{n_{re}^2\} + E\{n_{im}^2\} = V\{n_{re}\} + V\{n_{im}\}$$

$$V\{n\} = \frac{1}{6} \cdot \left(\frac{1}{2^{W_d}}\right)^2 = \sigma_{BF}^2$$

The analysis are for a single radix-2 DIF PE. This result can be used for the error analysis in the FFT. Consider the error in only on output node in the SFG in Figure 2.1. The error in that node is then the summation over all stages in the binary tree that is formed with that particular output node as root. Since safe scaling is used each error from a previous node is divided by 2 before it is added to the next stage. By propagating the error from the input to the output through the radix-2 PEs and their safe scaling, the noise variance for an *N*-point FFT [1] can be written as

$$\sigma_{FFT}^2 = \sigma_{BF}^2 \cdot \left(1 \cdot 2^2 + 2 \cdot 1^2 + 4 \cdot \left(\frac{1}{2}\right)^2 + \ldots + 2^{log_2(N)-1} \cdot \left(\frac{1}{2^{log_2(N)}}\right)^2\right) \qquad \text{(Eq 4.7)}$$

$$\sigma_{FFT}^2 = \sigma_{BF}^2 \cdot 2^2 \cdot \sum_{i=0}^{log_2(N)-1} \frac{1}{2^i} = \sigma_{BF}^2 \cdot 2^3 \cdot (1 - 2^{-log_2(N)}) = 8 \cdot \sigma_{BF}^2 \cdot \left(1 - \frac{1}{N}\right)$$

4.3.3 Radix-*r* Quantization

The noise analysis of radix-*r* DIF quantization is similar to the radix-2 DIF quantization analysis, [1]. For the radix-*r* FFT the variance would be in the following way

$$\sigma_{FFT}^2 = \sigma_{BF}^2 \cdot r^2 \cdot \left(1 + \frac{1}{r} + \ldots + \frac{1}{r^{log_r(N)-1}}\right) = \sigma_{BF}^2 \cdot \frac{r^3}{r-1} \cdot \left(1 - \frac{1}{N}\right) \qquad \text{(Eq 4.8)}$$

Equation 4.8 shows that the noise is larger for higher radix implementations, even though it has fewer butterfly stages.

5 Implementation Choices

5.1 Introduction

The first part of the master's thesis report is only a summary of different approaches to the FFT problem. This chapter is going to explain the choice of an adequate algorithm, architecture, and so on, for the area in which the FFT processor is going to be used. A lot of trade-offs and choices will be explained. The chosen architecture in this section is the one going to be implemented later in the project.

5.2 Algorithm Choice

There are no restrictions on the algorithm choice, except that the algorithm should be able to compute the FFT lengths from 1k to 8k. When selecting algorithm, the goal of an architectural and easy understandable design have to be considered.

The radix-2 FFT algorithm has many good features. For example, it has low quantization noise level, and it is also easily parameterizable to the different FFT lengths.

Radix-r does not seem to be as good a choice as the radix-2 one, because it has higher quantization noise, and that it is not as easily parameterizable to the different FFT lengths. To be able to parameterize this algorithm to the general power-of-2 FFT lengths, different radix-r stages have to be used in the pipeline, resulting in a mixed radix implementation.

Since minimizing the number of multipliers is important, a good choice of algorithm would be the split radix one. It has a lower number of multipliers than all the above ones, but this algorithm results in a complex design, which will be harder to parameterize. The control of this type of processor would also be more complex.

The radix-2^2 algorithm is the most attractive algorithm. It can be thought of as a radix-4 algorithm with radix-2 building blocks. It has low number of multipliers, simple control structure and architecture.

Prime factor algorithms can not be used, because the right FFT lengths can not be calculated using this algorithm.

These are the reasons that the radix-2^2 FFT algorithm is going to be used in the FFT implementation.

5.3 Architecture Choice

The choice of the architecture is easier, it has to be a pipelined architecture. What is left to decide is what kind of commutators to be used in the architecture, MDC, SDF or SDC. In the implementation the SDF type of commutator will be used, because it has a smaller memory requirement than the other commutators.

The radix-2^2 architecture behaves a bit like the radix-4 architecture, this calls for two different architectures later, to be able to implement all the different FFT lengths needed. The first architecture is for power-of-4 FFT lengths, i.e. 1k and 4k FFTs, and the second architecture is for 2k and 8k FFTs. The latter lengths can easily be created from the former ones by adding a radix-2 stage at either the input or the output of the FFT.

6 Radix-2^2 FFTs

6.1 Introduction

This chapter will describe the radix-2^2 FFTs in detail. The mathematical background to the algorithm and the architecture, will be discussed.

6.2 Algorithm

The derivation of the radix-2^2 FFT algorithm starts with a substitution with a 3-dimensional index map, [2]. The index *n* and *k* in Equation 2.1 can be expressed as

$$n = \left\langle \frac{N}{2}n_1 + \frac{N}{4}n_2 + n_3 \right\rangle_N \qquad \text{(Eq 6.1)}$$

$$k = \langle k_1 + 2k_2 + 4k_3 \rangle_N$$

When the above substitutions are applied to DFT definition, the definition can be rewritten as

(Eq 6.2)

$$X(k_1 + 2k_2 + 4k_3) = \sum_{n_3=0}^{\frac{N}{4}-1} \sum_{n_2=0}^{1} \sum_{n_1=0}^{1} x\left(\frac{N}{2}n_1 + \frac{N}{4}n_2 + n_3\right) \cdot W_N^{\left(\frac{N}{2}n_1 + \frac{N}{4}n_2 + n_3\right)\cdot(k_1 + 2k_2 + 4k_3)}$$

$$= \sum_{n_3=0}^{\frac{N}{4}-1} \sum_{n_2=0}^{1} \left\{ B_{\frac{N}{2}}^{k_1}\left(\frac{N}{4}n_2 + n_3\right) \cdot W_N^{\left(\frac{N}{4}n_2 + n_3\right)k_1} \right\} \cdot W_N^{\left(\frac{N}{4}n_2 + n_3\right)\cdot(2k_2 + 4k_3)}$$

Where

$$B_{\frac{N}{2}}^{k_1}\left(\frac{N}{4}n_2 + n_3\right) = x\left(\frac{N}{4}n_2 + n_3\right) + (-1)^{k_1} \cdot x\left(\frac{N}{4}n_2 + n_3 + \frac{N}{2}\right) \quad \text{(Eq 6.3)}$$

is a general radix-2 butterfly.

Now, the two twiddle factors in Equation 6.2 can be rewritten as

$$\begin{aligned} W_N^{\left(\frac{N}{4}n_2 + n_3\right) \cdot (k_1 + 2k_2 + 4k_3)} &= W_N^{Nn_2k_3} W_N^{\frac{N}{4}n_2(k_1 + 2k_2)} W_N^{n_3(k_1 + 2k_2)} W_N^{4n_3k_3} \\ &= (-j)^{n_2(k_1 + 2k_2)} W_N^{n_3(k_1 + 2k_2)} W_N^{4n_3k_3} \end{aligned} \quad \text{(Eq 6.4)}$$

Observe that the last twiddle factor in the above Equation 6.4 can be rewritten.

$$W_N^{4n_3k_3} = e^{\frac{-j2\pi}{N} \cdot 4n_3k_3} = e^{\frac{-j2\pi}{4N} \cdot n_3k_3} = W_{\frac{N}{4}}^{n_3k_3} \quad \text{(Eq 6.5)}$$

Insert Equation 6.5 and Equation 6.4 in Equation 6.2, and expand the summation over n_2. The result is a DFT definition with four times shorter FFT length.

$$X(k_1 + 2k_2 + 4k_3) = \sum_{n_3 = 0}^{\frac{N}{4} - 1} [H(k_1, k_2, n_3) W_N^{n_3(k_1 + 2k_2)}] W_{\frac{N}{4}}^{n_3k_3} \quad \text{(Eq 6.6)}$$

The result is that the butterflies have the following structure. The BF2II butterfly takes the input from two BF2I butterflies.

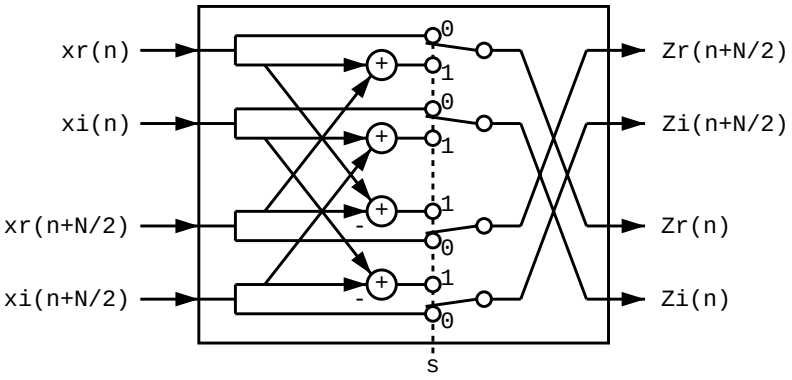
$$H(k_1, k_2, n_3) = \left[x(n_3) + (-1)^{k_1} x\left(n_3 + \frac{N}{2}\right)\right] + (-j)^{(k_1 + 2k_2)}\left[x\left(n_3 + \frac{N}{4}\right) + (-1)^{k_1} x\left(n_3 + \frac{3N}{4}\right)\right] \quad \text{(Eq 6.7)}$$

These calculations are for the first radix-2^2 butterfly, or its components the BF2I and BF2II butterflies. The BF2I butterfly is the one represented by the formulas in brackets in Equation 6.7 and the BF2II butterfly is the outer computation in the same equation. The complete radix-2^2 algorithm is derived by applying this procedure recursively.

6.3 Architecture

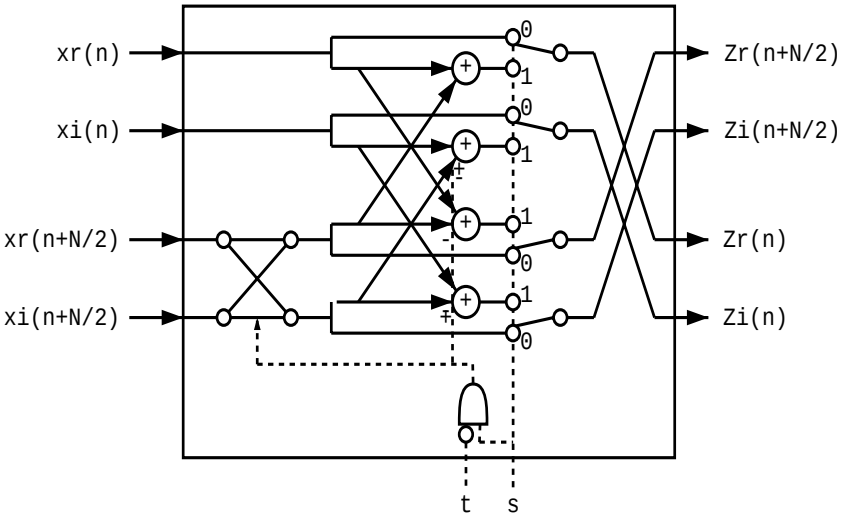
The first butterfly, the BF2I, in the radix-2^2 butterfly has the following architecture.

Figure 6.1: BF2I DIF butterfly architecture.



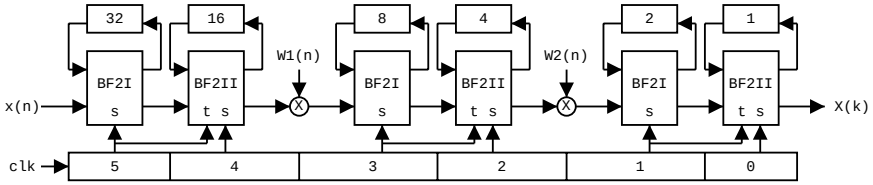
The second butterfly, the BF2II, has the architecture seen in the figure below. The BF2I butterfly is a radix-2 butterfly, whereas the BF2II butterfly basically is a radix-2 butterfly but with trivial twiddle factor multiplications.

Figure 6.2: BF2II DIF butterfly architecture.



A radix-2^2 SDF FFT architecture with these radix butterfly elements plus multipliers is shown in Figure 6.3. This architecture uses the same amount of non-trivial complex multipliers as the radix-4 architecture, but retains the simple radix-2 architecture. Another advantage is that the control structure is simple for this implementation, only a binary counter. The block in the feedback loop is a FIFO buffer, the number indicates the number of complex samples it can store.

Figure 6.3: Architecture of a 64-point radix-2^2 SDF FFT.



6.4 Numerical Effects

Numerical effects in the radix-2^2 algorithm is exactly the same as in the radix-2 algorithm, because it has the same butterfly structure, but with fewer multipliers. For a description of the numerical effect for the radix-2^2 algorithm, see the radix-2 investigations in chapter 4 Numerical Effects.

7 FFT Design

7.1 Introduction

This chapter discusses the design of an FFT processor. Different abstraction levels, block division, trade-offs, simulations, testing, etc. are things discussed in more detail.

In general, the design in this project should be done with top-down methodology in small refining steps. The first model will be built in Matlab and the final model will be FPGA synthesizable VHDL code. The design process will go from the former to the latter in several small design steps, i.e. only small changes will be introduces in the model in each step. These smaller design steps will hopefully lead to a more predictable design process and smaller amount of errors introduced when refining. At some point in the design process the Matlab model has to be converted to VHDL description, but it is hard to know in advance when the best time for this conversion will be.

The models should be implemented with a good hierarchical architecture. This leads to a more reusable and easier understanding of the final implementation.

7.2 Matlab Design

The design of the FFT processor begins with the design of a simple functional model in Matlab. The advantage of starting the design process in Matlab is that Matlab offers a high level programming language and a good interface for testing. This means that in a short time a lot of different models can be tested.

The first Matlab model was designed with a bottom-up methodology at a high level of abstraction. Actually a top-down methodology should have been used, but it seemed like a better solution to do the first model in this way, because a good description of the algorithm and architecture was available [2]. Some things in the algorithm were left out, which later caused problems that delayed the project for around two weeks. The bottom-up methodology was only for the creation of the first model and from that point and onwards a top-down methodology was used.

7.2.1 Problems and Solutions

All the different blocks were easily implemented, except the twiddle-factor generating block. The other blocks where easy to understand and were described in detail in the paper [2], but the twiddle-factor block was almost completely left out. Only the deduction of the algorithm gave a hint about the functionality of the block. After testing a lot of different models to get the FFT to work, the 16-point FFT finally worked correct, i.e. two radix-2^2 stages. To get it to work for longer FFTs was a real hard problem, because this was the part where some descriptions were left out. Finally it was solved through more studying of the FFT formulas, to understand them better.

7.3 Matlab Simulations

Matlab simulations are an important part in the design process. The simulation shows if the models developed are functionally correct. Not only the final FFT model were tested through simulations, but also all the different blocks in the processor were tested separately.

Most simulations were done to get an estimation of the size of the error in the output. The error depends on the data-widths in the processor and the number of steps (depends on the FFT length) in the pipeline. A lot of these simulations were carried out to test different data-width optimization techniques. Different optimization techniques are discussed in Section 9.2 on page 49.

Simulations were also done to compare two models against each other, to validate that they are functionally equivalent.

7.4 VHDL Design

The VHDL design started when the model of the parameterizable FFT processor were decided to be correct after a lot of simulations. The step from Matlab to VHDL should be as small as possible. The models should have the same blocks and their implementation should be the same. In VHDL it is possible to write functional models so the Matlab and the VHDL model should not differ so much.

The design environment that was used was Emacs for VHDL text-editing, Vcom for VHDL compiling and Vsim for VHDL simulation. There is a tool for graphical representation off block structures, but the structure is easier to understand with a total text representation, at least in this project with a lot of parameterization.

7.4.1 Problem 1 and Solution - Abstraction 1

In the Matlab model signals are described by complex variables. The IEEE library have support for complex signals, but only for floating point representation and not for the signed fractional two's complement representation. Hence, the first refinement between Matlab and VHDL was to separate the complex signals into two signals, one holding the real value and the other holding the imaginary value. This didn't cause much problems, for one reason the abstraction level was almost the same, and for another reason some of these models were already implemented in Matlab.

One problem that the division of the complex signals caused was that it increased the number of signals in the blocks almost by a factor of two, increasing the block complexity. Increasing the block complexity decreases the ease of understanding it.

The first approach to solve this problem was to create an array with two elements of a std_logic_vector, to create an abstraction of complex signals. However, it was impossible two make a construct in this way. The code wouldn't compile because the array elements have to be constrained before compile-time, and the word length couldn't therefore be parameterized.

The second approach also used arrays. The approach used an array with word length number of std_logic_vector(1 downto 0). This construct is compilable. The problem with it is that slices couldn't be used, a special function would have to be written to extract the information. The code would be easy to read, and it might even be synthesizable, but the testing will be more complicated. The std_logic_vector in this case only stores two bits, one for the real value and one for the imaginary value. In the test bench it will therefore not be easy to read the signal values.

The third approach was to create an array of std_logic. The array would have a size of 2 x the word length. This is a good way but it has some drawbacks. Both dimensions in the declaration of the array have to be unconstrained, the best way would be to be able to set one dimension to 2 and the other as a parameter. The drawbacks is that in the declaration of a signal two dimensions have to be given, one for the word length, and one for declaring the real and imaginary dimension (always 2). The declaration of signals in this way only makes the code a bit less readable.

The final approach, the one that was used in the implementation, abandoned the abstraction of signals. The reason was that there was another good solution. An extra layer in the block structure was added, decreasing the number of signals in each block. This gave VHDL-files of reasonable complexity and length.

7.4.2 Problem 2 and Solution - Object Orientation

The abstraction problem described above could have been solved with an object oriented variant of VHDL. There are some attempts the create this functionality with an extra layer on top of VHDL. One solution had a preprocessing stage, i.e. digital structures were written in a language different from VHDL. To synthesis this, the code first had to be compiled into VHDL-code, the rest of the steps are the usual VHDL-synthesis steps.

This solution wasn't used because VHDL had to be used according to the requirements, and because most people don't understand the code of the extra layer. The lack of understanding would limit the use of the code in the future.

When writing normal non-parameterized VHDL-descriptions the benefit of object orientation might not be as large as for highly parameterized systems like the FFTs considered in this project.

7.4.3 Problem 3 and Solution - Control Block

The control problem arose in the synchronization of control signals and data signals, i.e. that the right data should be available in the right state of control signals.

To get the FFT processor to work, shimming delays had to be added between each radix-2^2 stage and between the two butterfly elements inside the radix-2^2 stage. The result was that more HW was needed and that the latency between input and output frames increased. The latency is not a big problem, but the extra HW will increase the die size and the power consumption.

This problem could be solved in two ways, either changing the control unit or creating a system consisting of locally synchronous blocks communicating asynchronously (GALS). The first choice of keeping the FFT completely synchronous would increase the complexity of the control unit, resulting in a system that is harder to understand. The second choice would only have a slightly different control structure, but very similar to the original one. The blocks would also be more separated from each other functionally, which could be a good property when improving the design later in the future. These pros and cons lead to the implementation of the FFT processor as a GALS-system.

7.5 Design for Test

Design for test is a way to speed up the testing of manufactured chips. This design method is used to find fabrication faults in the chips, e.g. dust in the printed chip causing logical errors, not design errors. A measurement often used in this context is fault coverage, which often means the coverage of stuck-at faults. A fault coverage of e.g. 95% means that 95% of the die area of the chip was not corrupted during the fabrication process.

Design for test have not yet been considered, due to the early stage of the project.

7.6 VHDL Simulations

Test bench code-skeletons were produced for combinatoric, synchronous and asynchronous parts. These test benches could easily be altered to a specific test bench for a block. Input and output from the test benches were read and written to test files. These files could then later be read into Matlab where the VHDL simulations could be compared with the Matlab simulations.

To ease the interfacing between Matlab and VHDL simulation a set of Matlab functions were developed. These functions handled the generation of test data and the reading and writing of binary data to the test files.

7.7 Synchronous or Asynchronous Design

Both synchronous design and asynchronous design have advantages and disadvantages. The reason for choosing asynchronous design (GALS) in this project can be found in Section 7.4.3 on page 35. More about asynchronous circuits and the design of these can be found in Section 8 on page 41.

7.8 Testing

Testing is an important part of the project and testing is done on all levels. This section will outline the testing strategy of this project. Previous sections have been describing simulations, which also is a form for testing, but this section looks into this area more thoroughly.

7.8.1 Random Testing

Random testing is the most frequently used method. Random sequences are generated by Matlab, written to the test files and read by the VHDL test benches. This type of testing is quick to use because test sequences are generated automatically, and it is also adequate in the area of FFT's because input signals often seem to be randomly distributed.

7.8.2 Corner Testing

Random testing is good, but some things is hard to detect, e.g. corner cases. Corner cases are input sequences that the designer or tester think can cause errors, e.g. overflows in adders and multipliers. In the FFT area a corner case could be an input sequence of only maximum and minimum input values.

In this project corner testing is mostly done to check that the safe scaling works as in should, resulting in no overflows which would cause errors in the output.

7.8.3 Block Testing

Block testing is the lowest level of testing. Before a block is built out of other sub-block, the sub-blocks are run through a block test to ensure that each sub-block is verified. When it is known that all sub-blocks work correctly it can be assumed that if the block errors it is due to the interconnection of sub-blocks. This method limits the area where the error is and will save a lot of debugging time.

7.8.4 Golden Model Testing

Golden model testing is the best way to ensure that a model is working as it should. The *Golden model* is a model that is known to be working correctly, which new implementations of the same functionality could be compared against.

In the case of this project the golden model is the built-in FFT function in Matlab. The Matlab function can of course only be used when testing the complete FFT, and not really the sub-blocks of the system. Since the Matlab model is thoroughly tested and it is assumed to be correct, the sub-blocks can be used as golden models for the testing of later designs.

7.8.5 FPGA Testing

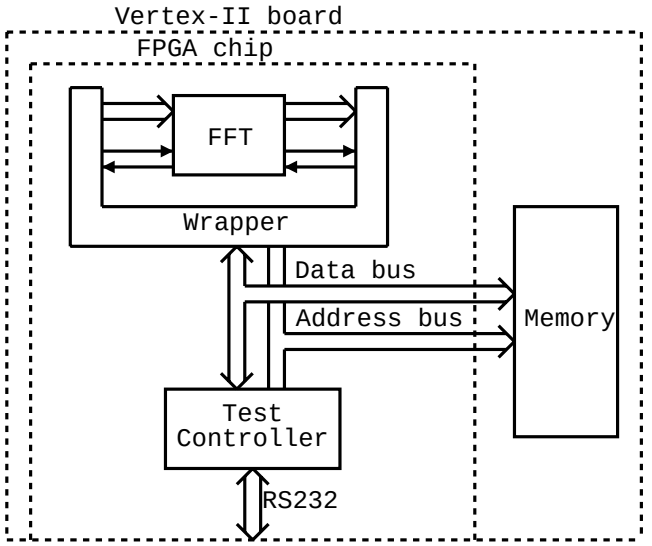
FPGA tests is the final step in the testing process. Doing simulations in computer software is time consuming, and it is therefore difficult to run large test vectors. VHDL code synthesized to an FPGA will speed up the test a lot, probably several orders of magnitude.

The Virtex-II V2MB1000 Development Board was used for the FPGA tests. The choice to use this board is that it can handle large designs. The development board has a lot of sockets for interfacing with other components, i.e. RS232, parallel input, ethernet and on board switches.

To do tests in real-time, test vectors have to be sent and received from the FPGA in full speed. This requires a high bandwidth through one of the FPGA board interfaces, since all test vectors cannot be stored on the FPGA board. These interfaces would take to much time to implement to fit the time-plan of this thesis project, hence a simpler test is required and therefore the real-time requirement is dropped.

The full speed test can be done by loading all test vectors into a memory on-board, run the simulation in full-speed while writing the output to memory, and finally reading out the FFT output from the memory. See the block schematic of the test bench in the figure below.

Figure 7.1: Vertex-II asynchronous FFT test bench.

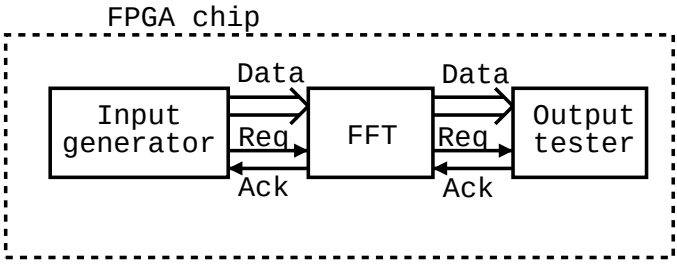


To test the design with this block structure would be possible, but even for this there is not enough time left in the project to finish the test. An interface have to be written for the memory and the RS232 port, as well as the code for the wrapper and test controller, and a program to communicate with the test board. To design all this would prolong the project time far beyond the limit.

The RS232 port seems to be the easiest way to interface with the Virtex-II board, so the other solutions using other ports would also extend the project beyond limits. Hence, another way of testing is required.

The final test bench was completely embedded on the FPGA chip. Since the FPGA chip does not have a large memory (ROMs and RAMs) capacity, the test vectors have to be small. Instead of testing a 1024-point FFT a smaller FFT was tested. A 16-point FFT was selected because it is the smallest FFT processor in this project that includes all components, i.e. the two different butterflies, prescaling, final scaling and the twiddle factor multipliers.

Figure 7.2: The implemented FPGA test.



The Input generator was implemented with a ROM memory, a counter and an asynchronous wrapper. The Output tester was implemented in a similar way, but with an extra block that compared the received data with the expected data. A difference between the received and the expected data would trigger the test bench into an error mode, which would light a diode on the FPGA board.

The result of the test showed that it was possible to synthesize the FFT processor to an FPGA.

7.9 Synthesis

The VHDL-code written in this project should be synthesizable. For the synthesis of the code two programs were used, LeonardoSpectrum and Xilinx Design Manager. The synchronous parts were easily synthesized, but the asynchronous ones caused a lot of problems, Section 8.7 on page 46.

7.10 Meetings

Meetings were held regularly. Every meeting had a written protocol and a minutes were written directly after the meeting. The minutes were then sent by e-mail to the examiner and the supervisor.

In the beginning the meetings were held to define the limitations and directions of the project, and later the meetings mostly described the progress in the work.

The meetings helped a lot in the beginning, because it defined clearly what was going to be done. If something was forgotten, the corresponding minutes could be read to find the answer, if not, it was included in the protocol for the next meeting.

8 Asynchronous Design

8.1 Introduction

An asynchronous design methodology was used in this master's thesis project to solve the problem with the control signal and data synchronization. Asynchronous design has many interesting properties, but the reason for using it was to test if it could solve the control problem.

This chapter will introduce asynchronous circuits and it will also describe the asynchronous design process used in this project.

8.2 Asynchronous Circuits

Asynchronous circuits are a big area of research and only small parts of it will be used in this project, the theory will therefore only be briefly outlined. For the interested reader more material on asynchronous circuits can be found in [3].

Asynchronous circuits have many advantages over synchronous ones, like

- Performance of an asynchronous system depends on the average case latency, not the worst case latency as in synchronous circuits.
- Global clock timing problems are avoided.
- The power consumption can be lower in asynchronous circuits, despite the fact that they require more HW.

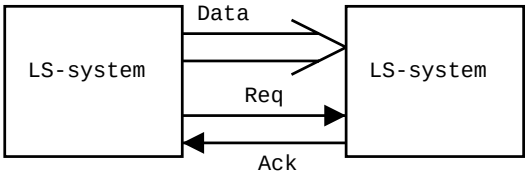
- Asynchronous systems are more robust against temperature and supply voltage variations.
- Lower electrical interference with other components, due to the lack of clock harmonics in the emission spectra.

No global clock is used in asynchronous circuits, instead some form of handshaking is used in the communication between systems. Two examples of handshaking protocols are 2-phase and 4-phase handshaking. In this project 4-phase handshaking will be used, because these interfacing blocks already have been implemented in VHDL.

8.3 GALS

GALS, abbreviation of Globally Asynchronous Locally Synchronous, are a small subset of asynchronous systems. These systems are not completely asynchronous, they consist of synchronous sub-systems communication asynchronously. In Figure 8.1 the LS-system is a locally synchronous system, *Req* is short for request and *Ack* is short for acknowledge. *Req* and *Ack* performs the handshaking. In this project a push communication channel will be used, which means that the producer of data initiates the handshaking.

Figure 8.1: GALS asynchronous communication.



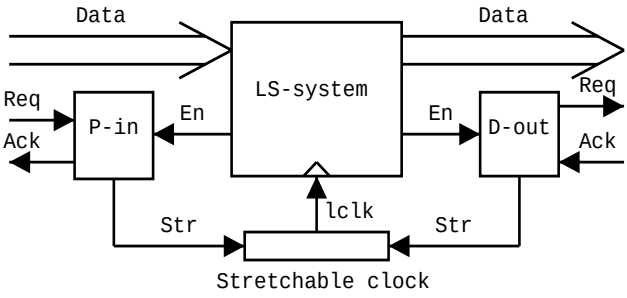
A 4-phase handshaking cycle is performed in the following way (*Req*+ means *Req* goes high): *Req*+, *Ack*+, *Req*- and finally *Ack*-. Data should be valid between *Req*+ and *Ack*-, but is often sampled on the *Ack*+ edge.

GALS combine some of the benefits from synchronous circuits with the benefits from asynchronous ones. Since the local parts are synchronous they can be designed the same way as before using available design tools. Globally the system is asynchronous, which removes timing problems for global signals, which is an increasing problem when designs are getting larger and faster.

8.3.1 Asynchronous Wrappers

An asynchronous wrapper is used for the handshaking between two modules. The wrapper consists of three components, a stretchable clock generator, a demand-port (D-port) and a poll-port (P-port). Connected according to the figure below they implement a 4-phase push-channel, [3].

Figure 8.2: Asynchronous wrapper components.



The *En* signal is a control signal from the LS-system, that controls the input and output of the system. *En* on the input controls when the LS-system is ready to receive data, and *En* on the output controls when the LS-system is ready to send data. The *Str* signal stops the clock if no input data is available or the receiving block is busy. The *lclk* signal is the clock for the LS-system.

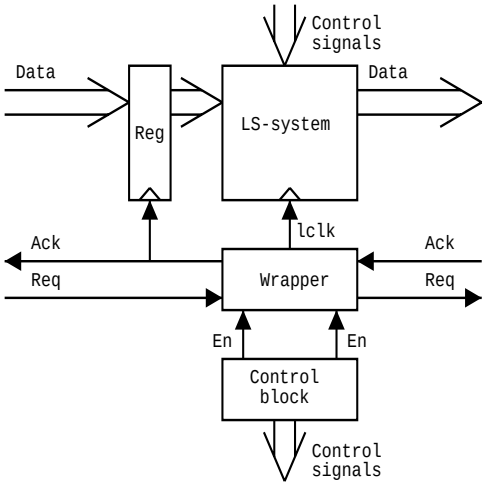
8.3.2 Enable generation

The enable signal can be generated in a lot of different ways. Though, for this problem an easy solution is possible. Data flows through the components continuously and the same amount of processing is needed for each sample. The system both needs data and can send data each clock cycle. Though, the output is delayed compared to the input, i.e. the output enable generation has to be delayed compared to the input enable generation. The result is that the enable control can be a separate block built up mostly from a counter.

8.4 Design Automation

To be able to quickly convert the available synchronous implementation to a GALS one, the design was automated. The wrapper was implemented as one component, a VHDL code skeleton was created for one LS-system with an asynchronous wrapper and data registers, and test benches for these systems were created. The general structure of an asynchronous component in this project uses the architecture in the figure below.

Figure 8.3: General architecture of asynchronous components.



The skeleton according to this structure is easy to modify to wrap any LS-system of the considered type.

The only difference between two different asynchronous blocks are the control block, which of course depends on what LS-system is being wrapped. In the synchronous implementation the control block is global, whereas when the system is divided into several asynchronous block each block have to have its own local control block. The local control blocks were similarly implemented as the global one.

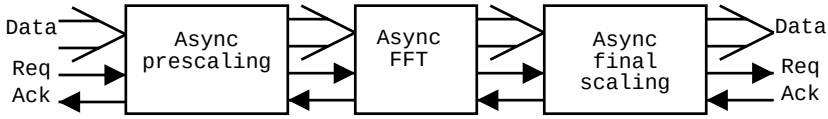
The skeleton above only works for a special type of architectures. Firstly, input is only received from one block and sent to another block. Secondly, the structure implements a push-channel type of communication. Though, it is easy to change it to a pull-channel type by reversing the Req

and Ack signals, and swap the D-port for a P-port and the P-port for a D-port in the wrapper.

8.5 Asynchronous FFT Architecture

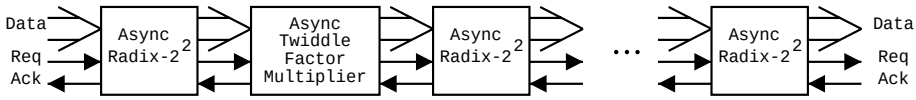
The asynchronous FFT is built up by connecting the wrapped LS-systems, which is illustrated in the figure below.

Figure 8.4: The architecture of the asynchronous FFT, including scaling.



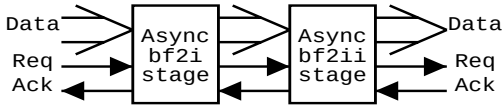
The number of stages are determined by the FFT transform length. Figure 8.5 shows the FFT structure with a power-of-4 transform length. In the case a 2 times power-of-4 length is wanted, an extra butterfly stage and twiddle factor multiplier stage have to be added on the input side of the processor. The reason for choosing the input side and not the output side, which also is possible, is that a layout could be done of the first five radix-2^2 (1024-point FFT) stages and the twiddle factor stages in between. This layout would not change in the case when extra butterfly stages are added on the input side.

Figure 8.5: The architecture of the asynchronous FFT.



Decomposing the radix-2^2 blocks into two different asynchronous units removed the timing problem of the internal control signals in the stage.

Figure 8.6: The architecture of the radix-2^2 block.



The asynchronous butterfly stages, the twiddle factor multiplying stages and the scaling stages were designed according to the methodology described in Section 8.4 on page 44.

8.6 Testing

The testing of the asynchronous circuits are similar to the testing of the synchronous ones. A skeleton test bench were implemented that allowed the use of the same input and output files as for the testing of the synchronous design.

8.7 Synthesis

The synthesis process of the asynchronous parts required a lot of time. In the VHDL simulations the asynchronous blocks were functional, these are not synthesizable, but synthesizable versions of these blocks were available. These blocks, since they are asynchronous, need redundant logic to be functionally correct and redundant logic is removed by the synthesis tools to optimize the design. It took a lot of time to find out that this was the problem when the FPGA simulations didn't work, since the FPGA does not offer any means of seeing what actually is happening inside the chip.

The synthesis tools have a possibility to see the generated components and the problem with the removed logic was found. The next time consuming problem was to find out how to prevent the synthesis tools from removing logic in certain components. The solution was to use an attribute (keep) in the VHDL code and also setting an attribute (PRESERVE_SIGNAL) in the tcl-script used in the synthesis by LeonardoSpectrum.

8.8 Summary of GALS Design

To design a GALS-system instead of a completely synchronous system proved to be a quick and efficient way to solve the problem with the timing of data and control signals. All unnecessary buffers in the processor could be removed in one week of work, not including the studying of asynchronous circuits, which took only a few days.

The design principle is easy to use, not much understanding is needed of the asynchronous parts, only the use of them have to be learnt. With a package including the wrappers etc., a designer unexperienced with GALS could get started designing these systems in a couple of days.

The only problem encountered with the GALS design was the synthesis, but when the process to do the synthesis of the asynchronous parts have been learned, this design step will not cause much problems.

9 Future Work

9.1 Introduction

In this chapter the future work of the project will be discussed. The chapter suggests where improvements are possible or important, and interesting continuations of the project.

9.2 Word Length Optimization

Word length optimization is one of the most important future works. Since the FFT processor is parameterized, the different parameters for specific applications should be easy to find using some optimization algorithm. A few optimization test have been done, and this section will outline them, plus describe some areas where more work could be done.

9.2.1 General

In general, optimizing a HW structure is a trade-off between precision, power consumption, die size, and so on. How should these parameters be weighed against each other? This depends of course on the application. When the application is know the needed precision of the output is known. Since the precision is non-negotiable there is a fixed constraint on the precision, an optimized solution could therefore not have worse precision than this constraint. Hence, the optimization problem is divided into two parts: getting good enough precision, and maximizing the utility of the other parameters, as power consumption and so on. These parameters are weighted together to a utility value (the importance between parameters depends on the application).

9.2.2 Gradient Search

The test on gradient search that has been done uses a vector of word length parameters. One at a time each parameter is changed one step and the utility and output error are calculated for each change. This results in a vector telling the importance (the gradient) of each parameter, both in utility and precision. If there are solutions with better precision than the constraint the one with the highest utility is chosen for the next iteration, and if no solution have a good enough precision, the solution with best precision is chosen for the next iteration.

It was hard to test the convergence of this algorithm, mostly due to the lack of a relevant function to calculate the utility of a set of parameters. One way to solve this is suggested in the next section.

9.2.3 Utility Function

One way on solving the problem to estimate the utility function is to synthesize the butterflies, ROMs, RAMs and multipliers with different word lengths, and measure the power consumption, the die area, and so on. Measurements does not have to be made for all possible different word lengths, because the ones not measured could be estimated with some kind of interpolation.

The problem is still how to weigh these parameters against each other, but with this method the solution is one step closer, since at least die size, power consumption, and so on have to be estimated.

The synthesis work for the different word lengths have not been done. It is left for the future.

9.3 VLSI Layout

The work in this project ends in FPGA synthesizable code. FPGAs are good in many ways. They have a short development cycle, because layouts does not have to be done and no layouts have to be sent for fabrication. Though, they cannot compete with VLSI layouts in speed and power consumption, and when it comes to large series of components the FPGAs can not compete with the prize.

Mostly due to the better performance, in speed and power consumption, that a VLSI design gives, there could be an interest in doing a VLSI layout of this FFT processor. It could of course also be interesting to see if the processor, considering the GALS design, is easy to layout into a working chip.

9.4 VLSI Layout of Asynchronous Parts

The GALS design of this project is interesting. The high level design of these circuits turned out to be simple. It could therefor be interesting to learn more about the design flow of these kinds of circuits all the way down to VLSI layout. Hopefully the design on that level will be as easy as the high level design. The reason that this FFT processor could be a good test example is that it is a GALS design, that could be used in a real-world application.

9.5 Completely Asynchronous Design

The design of the FFT processor is so far of the GALS type, but there is no reason that the LS-systems have to be kept synchronous. These systems could also be broken down into smaller systems, which also could be of the GALS type, or even completely asynchronous.

9.6 Design for Test

DFT, or design for test, is increasing in importance due to the increasing size of designs. Adding design for test methodology to the project could be interesting later on in the project before a VLSI layout is going to be done.

9.7 Twiddle Factor Memory Reduction

A twiddle factor memory reduction is possible. The reduction that can be accomplished is reducing the ROM memories by a factor of 8. The reduction of the ROM memories will decrease the die size of the design, but will probably increase the power consumption because the factors in the ROM have to be changed in some way to match the previous twiddle factors.

9.8 Commutators Implemented with RAM

The current implementation of the commutators uses a FIFO buffer implementation. To use a RAM memory to implement the FIFO buffers will reduce the power consumption.

9.9 Unscrambler

An unscrambler could be a useful block to implement. It is not always required to get the output data in natural order, but when it is an unscrambler is needed. In the current solution the output arrives in bit-reversed order.

10 Summary

10.1 Conclusions

The goal with this master's thesis project was to learn more about the design of FFT processors, and designing a parameterized FFT processor. The processor has been designed and a design methodology with small design steps have been used successfully.

In the end of the project the design turned into an asynchronous design, leading to more interesting problems and solutions not expected from the beginning. The design of the GALS FFT taught a lot in this area, and a simple design methodology for this area has also been developed.

Finally the next section will go through the requirement stated early in the project, to check that they are fulfilled.

10.2 Follow-up of Requirements

The "Follow-up of Requirements" will go through the requirements stated in chapter 1 Introduction.

1. The transform length shall be able to vary between 1k and 8k samples in powers-of-2.

   This requirement is fulfilled. The result is even better than the requirements, the implemented architecture handles any frame length of a power-of-2 number, not only the ones between 1k and 8k.

2. The input signal shall be a continuous data stream.

This requirement is fulfilled. The implemented architecture handles a continuous data stream in natural order, and the output is received in bit-reversed order.

3. The input signal shall consist of only one continuous data stream.

   This requirement is fulfilled. The implemented architecture handles one and only one data stream.

4. The word length of the input and output signal shall be parameterizable. The internal word lengths shall also be parameterizable.

   This requirement is fulfilled. The implemented architecture is completely parameterized. Word lengths in butterfly stages and in all multipliers, as well as input and output, can be specified.

5. Safe scaling shall be used.

   This requirement is fulfilled. The implemented architecture uses safe scaling, using a scaling factor of 2.

6. Data shall be represented with two's complement format.

   This requirement is fulfilled. Data representation in the implemented architecture uses fractional two's complement representation.

7. The implemented architecture shall be pipelined.

   This requirement is fulfilled. The implemented architecture uses pipelining. Each butterfly and each complex multiplier is a step in the pipeline. More pipelining stages can of course be inserted, if needed.

11 Bibliography

[1] Torbjörn Widhe. Efficient Implementation of FFT Processing Elements. Thesis No. 619, Department of Electrical Engineering, Linköping University, Sweden. 2002.

[2] Sousheng He, and Mats Torkelsson. “A New Approach to Pipeline FFT Processor”. Department of Applied Electronics, Lund University, SWEDEN.

[3] Jens Muttersbach, et. al. Globally-Asynchronous Locally-Synchronous Architectures to Simplify the Design of On-Chip Systems. Integrated Systems Laboratory, Swiss Federal Institute of Technology, Zürich, Switzerland. 1999.

[4] Bevan M. Baas. An Approach to Low-Power, High-Performance FFT Processor design. Dissertation Department of Electrical Engineering, Stanford, USA. 1999.

[5] Shigenori Shimizu. Multi-Pipeline FFT Architecture. The transactions of the IEICE, vol. E 70, no. 6 June 1987.