Tampere University

Jukka Loukkaanhuhta

FFT-BASED CORRELATOR DESIGN FOR GNSS RECEIVER USING HIGH-LEVEL SYNTHESIS

Master of Science Thesis
Faculty of Information Technology and Communication Sciences
May 2021

ABSTRACT

Jukka Loukkaanhuhta: FFT-Based Correlator Design for GNSS Receiver Using High-Level Synthesis
Master of Science Thesis
Tampere University
Degree Programme in Electrical Engineering, MSc (Tech)
May 2021

New Global Navigation Satellite Systems (GNSS) pose new challenges to the receiver architecture. One of these challenges is the increasing length of a pseudo-random-noise (PRN) code sequence in the signal. As this code is used in the signal acquisition process to calculate correlation, the increasing length makes the correlation task more time-consuming. The traditional method of calculating the correlation has been in the time-domain, but a possibility for correlation in frequency-domain also exists.

In this thesis, the frequency-domain correlation option is explored by creating a correlator design, that utilizes Fast Fourier Transform (FFT) to speed up the transformation between time- and frequency-domains. This provides potential performance gain over the time-domain approach. The implementation of this design is done using High-Level Synthesis tools, which saves some time on the coding part and makes it possible to focus more on the algorithm side of the process.

The presented design is then compared with a pre-existing time-domain reference model. Analysis is done on the accuracy of the correlation results, as well as on the performance and area of the design. The actual presented implementation is sub-optimal and doesn't quite achieve the performance of the reference model with the target clock frequency. Still, the collected results prove the potential speed improvement over the time-domain approach, especially on the higher code lengths. On the other hand, this performance comes with a price since the memory requirements make the FFT-based design larger than the time-domain version. Depending on the use case, the FFT-based correlator is proved to be an approach worthy of consideration.

Keywords: GNSS, global navigation satellite system, correlator, FFT, fast Fourier transform, radix-2 FFT, mixed-radix FFT, HLS, high-level synthesis

The originality of this thesis has been checked using the Turnitin OriginalityCheck service.

TIIVISTELMÄ

Jukka Loukkaanhuhta: FFT-pohjainen korrelaattoritoteutus GNSS vastaanottimeen korkean tason synteesiä hyödyntäen
Diplomityö
Tampereen yliopisto
Sähkötekniikan DI-tutkinto-ohjelma
Toukokuu 2021

Uudet satelliittipaikannusjärjestelmät aiheuttavat uusia haasteita vastaanottimille. Yksi näistä on seurausta satelliittien lähettämissä signaaleissa olevien koodisarjojen pituuksien kasvusta. Näitä koodeja käytetään satelliittien etsimisessä ja niiden avulla lasketaan korrelaatiota vastaanotettujen signaalien ja tunnettujen koodien välillä. Koodien pituuksien kasvu aiheuttaa korrelaation laskentaan käytetyn ajan kasvun. Tyypillisesti korrelaation laskenta on toteutettu aikatasossa, mutta se olisi mahdollista toteuttaa myös taajuustasossa.

Tässä työssä taajuustason korrelaatiota tutkitaan toteuttamalla korrelaattori, joka hyödyntää nopeaa Fourier-muunnosta (FFT) nopeuttamaan muunnosta aika- ja taajuustasojen välillä, mikä mahdollistaa paremman suorituskyvyn verrattuna aikatason korrelaatioon. Työ toteutetaan käyttämällä korkean tason synteesityökaluja, joiden avulla työhön käytetty aika on mahdollista kohdistaa paremmin algoritmipuoleen koodin kirjoittamisen sijasta.

Esiteltyä toteutusta verrataan olemassa olevaan aikatason korrelaattoriin ja analyysi kohdistuu korrelaatiotuloksien tarkkuuteen sekä toteutuksen nopeuteen ja kokoon. Tulokset paljastavat että työssä toteutettu korrelaattori jää suorituskyvyssä jälkeen referenssitoteutuksesta tavoitellulla kellotaajuudella, mutta FFT-pohjaisella korrelaatiolla on kuitenkin mahdollista toteuttaa nopeampi korrelaattori. Tämä suorituskyky korostuu erityisesti pitkillä koodinpituuksilla. Nopea toteutus kuitenkin tuo mukanaan suuremman pinta-alan sillä FFT-pohjainen lähestymistapa edellyttää suhteellisen suuria muistielementtejä. Tästä huolimatta FFT-pohjainen korrelaattori on varteenotettava vaihtoehto tietyissä käyttökohteissa.

Avainsanat: GNSS, satelliittipaikannusjärjestelmä, FFT, nopea Fourier-muunnos, radix-2 FFT, HLS, korkean tason synteesi

Tämän julkaisun alkuperäisyys on tarkastettu Turnitin OriginalityCheck -ohjelmalla.

PREFACE

This thesis was done for u-blox Espoo Oy. The work was done during summer and autumn of 2020 and the writing process continued during early 2021.

I would like to thank for the opportunity to do this thesis as it was a valuable experience for me. I would also like to thank my examiners for the guidance in this process as well as all my co-workers at u-blox for the help and support they offered me and for making working on this thesis a pleasant experience. Finally, I want to thank my friends and family for all the support during this process.

Tampere, 4th May 2021

Jukka Loukkaanhuhta

CONTENTS

1 Introduction . . . 1
- 1.1 Objectives . . . 1
- 1.2 Organization of the thesis . . . 2

2 GNSS Systems . . . 3
- 2.1 Satellite based positioning . . . 3
- 2.2 GNSS signal structure . . . 5
- 2.3 GNSS Receiver . . . 6
- 2.4 Signal acquisition using correlation . . . 7

3 Fast Fourier Transform based correlation . . . 9
- 3.1 Frequency-domain correlation . . . 9
- 3.2 FFT and Cooley-Tukey algorithm . . . 10
- 3.3 Inverse transform . . . 13
- 3.4 Radix and butterfly unit . . . 14
- 3.5 Zero padding and mixed radix . . . 16
- 3.6 In-place computation . . . 17
- 3.7 Hardware FFT . . . 19
  - 3.7.1 Pipelined FFT . . . 19
  - 3.7.2 Memory-based FFT . . . 20

4 Introduction to High-Level Synthesis . . . 22
- 4.1 High-level synthesis . . . 22
- 4.2 Catapult HLS . . . 23

5 Correlator using HLS . . . 24
- 5.1 From algorithmic model to hardware . . . 24
  - 5.1.1 FFT address generation . . . 24
  - 5.1.2 Bit reversal issue . . . 26
  - 5.1.3 C++ code structure . . . 27
  - 5.1.4 Bit-accurate data types and other HLS optimizations . . . 29
  - 5.1.5 Multiply and magnitude blocks . . . 30
  - 5.1.6 C++ test bench . . . 31
- 5.2 Final designs . . . 31
  - 5.2.1 Radix-2 . . . 32
  - 5.2.2 Mixed-radix-248 . . . 35

6 Results . . . 41

6.1 Verification methods 41
6.2 Results comparison 43
6.2.1 Correlation error 43
6.2.2 Performance 46
6.2.3 Design area 49
7 Conclusion and future Work 51
References 53

LIST OF PROGRAMS AND ALGORITHMS

3.1 Matlab code for FFT with three nested loops 17
5.1 C++ code for the correlator class 27
5.2 C++ code for radix-248 address generation 38

LIST OF SYMBOLS AND ABBREVIATIONS

ASIC	Application-Specific Integrated Circuit
CDMA	Code Division Multiple Access
CORDIC	COordinate Rotation DIgital Computer
DFT	Discrete Fourier Transform
DIF	Decimation-in-Frequency
DIT	Decimation-in-Time
FDMA	Frequency Division Multiple Access
FFT	Fast Fourier Transform
FIFO	First-In-First-Out
FPGA	Field-Programmable Gate Array
GLONASS	GLObal Navigation Satellite System
GNSS	Global Navigation Satellite System
GPS	Global Positioning System
HLS	High-Level Synthesis
IDFT	Inverse Discrete Fourier Transform
IFFT	Inverse Fast Fourier Transform
II	Initiation Interval
LSB	Least Significant Bit
MSB	Most Significant Bit
PRN	Pseudo-Random Noise
RMSE	Root-Mean-Square-Error
ROM	Read Only Memory
RTL	Register Transfer Level
SRAM	Static Random Access Memory
TOA	Time-of-Arrival
VHDL	Very High Speed Integrated Circuit Hardware Description Language

1 INTRODUCTION

Accurate navigation systems have become an increasingly common over the years and their applications can be found in such everyday items as mobile phones or cars. As the satellite based positioning systems as well as their accuracy have improved over the years, the technology has found its way into more and more applications. New possibilities however also mean new challenges and as new Global Navigation Satellite System (GNSS) technologies require more from the hardware, it is encouraged to look for new solutions to replace the older implementations.

1.1 Objectives

One of the challenges is caused by the increase in length of the code sequence transmitted in the GNSS signal. The code sequence is an integral part in positioning as it’s used to discover the satellites as well as to produce accurate positioning results. To achieve this, a receiver performs a correlation operation on the signal. This is traditionally done in time-domain, but it’s a rather time consuming process. As the code length increases, so does the computational load of this correlation process. An alternative for this time-domain correlation is to do it in frequency-domain instead.

In this thesis, the frequency-domain correlation is explored. Although the frequency-domain version makes correlation process more complex, it does provide theoretical benefits over the traditional approach. These benefits are due to the possibility of using Fast Fourier Transform (FFT) to speed up the transformation of the signal between time- and frequency-domains. The thesis work is focused on creating a hardware design of the correlator which utilizes FFT. This implementation is then evaluated against a pre-existing time-domain version of a correlator to provide results on the performance as well as to find out the important design aspects of this approach. Additional goal is also to provide some insight on High-Level Synthesis (HLS), which is used in the implementation process of the correlator.

1.2 Organization of the thesis

The chapter 2 focuses on giving a introduction to the GNSS systems and how they work. This chapter provides context to the correlator which is the focus of the thesis. Signals and basic structures of the receiver are presented and the purpose of the correlation process is explained as well as the correlation in time-domain.

Chapter 3 presents the correlation in frequency-domain and gives an idea about the functions and structure of the correlator. As the FFT is an important part of the frequency-domain correlation, the FFT algorithm is explained as well as the limitations and benefits of it. The chapter also gives an initial idea on how to implement the FFT algorithm with software as well as the options to do it in hardware.

The chapter 4 works as an introduction to high-level synthesis and brings up the advantages of using HLS as well as presents the tools used for this thesis. This is followed by chapter 5 which explains the actual implementation work of the correlator. This chapter also discusses the challenges in the design process and the choices that were made. In the end, the final correlator designs are presented.

Chapter 6 explains the methods for analyzing the designs and gathers the results for the implementations and compares them to the reference time-domain version. Finally the chapter 7 draws the conclusions from the results and presents some thoughts about the process. Suggestions for future work and improvements are also given.

2 GNSS SYSTEMS

This chapter takes a look into the basic principles of satellite navigation systems to get an understanding on how the GNSS systems work. It's also explained how the receiver is able to calculate its position.

2.1 Satellite based positioning

The term GNSS covers multiple different satellite navigation systems, for example the American Global Positioning System (GPS), Russian GLObal Navigation Satellite System (GLONASS), European Galileo and the Chinese BeiDou [1]. The GPS, being the oldest one of the aforementioned, is used as an example in this chapter.

In GNSS systems, the position is determined based on time-of-arrival (TOA). A satellite transmits a signal that a receiver detects. The signal includes timing information and if satellite and receiver clocks are synchronized to the same system time, the receiver is able to calculate the time it took for the signal to propagate from the satellite to the receiver. Based on the propagation time and speed of signal propagation, the receiver is able to calculate its distance to the satellite when their positions are known. When signals from multiple satellites are used, the position of the receiver can be determined [1].

As an example for two-dimensional positioning, first consider a satellite that transmits a signal which is detected by a receiver. Based on the timing information, the distance *R1* to the satellite can be determined. This means that the location of the receiver is somewhere on a circle of which center is on the location of the satellite and radius is *R1*. This is visualized in figure 2.1.

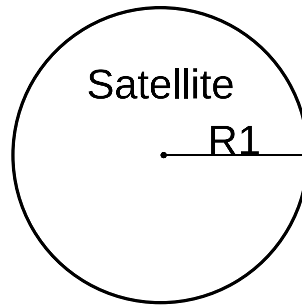

Figure 2.1. *Position based on one source*

When two satellites are used, the distances to those satellites are *R1* and *R2* respectively. When circles are drawn around the satellites with radii *R1* and *R2*, the circles intersect at two points, These two points are the possible locations for the receiver. This is shown in figure 2.2.

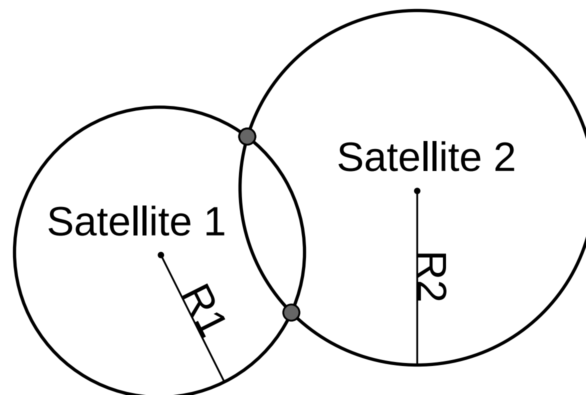

Figure 2.2. *Position based on two sources*

In similar fashion, when a third satellite is added, there's only one point where all of the circles intersect as shown in the figure 2.3. This is the location of the receiver. For three-dimensional positioning the circles would be replaced by spheres, but the principle remains the same.

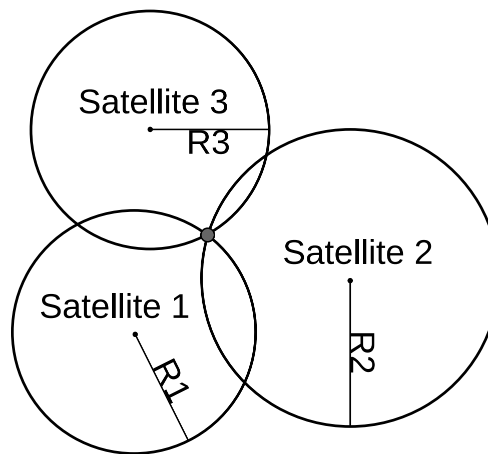

Figure 2.3. *Position based on three sources*

The previous scenario assumes that the receiver clock is perfectly synchronized with the satellite clocks. In reality this is not the case. The difference in clocks, as well as atmospheric effects on the signal propagation speed and interfering signals all cause error to the travel time measurements and thus affect the distance calculations and the accuracy of the estimated position [1].

2.2 GNSS signal structure

Satellites in GNSS systems transmit signals that make the positioning possible. These signals contain all the necessary information for accurate determination of location. The signals consist of different components and each of them has its own purpose.

The basis of a GNSS signal is the carrier, which is modulated to transmit the navigation message and the ranging code. The carrier signal typically uses frequencies in the L-band, which is in the range of 1 to 2 GHz. These frequencies offer good characteristics in regards to the required satellite transmitter power and attenuation caused by the atmosphere [1].

The ranging code, also known as a spreading code, consists of pseudo-random noise (PRN) sequence [2]. The length of the code depends on the GNSS system and can vary for example from 511 to 10230 bits or, as often called, chips to distinguish them from the navigation data bits.

The GNSS systems typically utilize Code Division Multiple Access (CDMA), which means that each satellite has a unique ranging code that is used to modulate the signal. This allows the satellites to broadcast on a common carrier frequency [1]. These codes are known to the receiver and can be used to separate the signals using correlation. This is

possible because the PRN codes are such that they have high auto-correlation and very low cross-correlation with each other. Because of this, it's possible to recover a weak signal among noise and stronger signals [3]. Another option would be to use the same code for all satellites, but have them to broadcast using different carrier frequencies as done in Frequency Division Multiple Access (FDMA) systems. [1]. The code sequence is used to determine the transmit time of the signal with high precision.

The Navigation data, which is modulated onto the signal, contains information about the satellites and the GNSS system. This includes various kinds of information about the satellite, but for the signal acquisition purpose, the interesting part is information used to determine the locations of the satellites. This can be used to decide which unique codes to search for during the acquisition if the approximate location of the receiver is known. The message also enables the offset between the receiver clock and the satellite clock to be determined, which is necessary for high accuracy positioning. The PRN code is used to determine the transmit time of the signal, but since the repetitive nature of the code sequence leaves ambiguity, it's necessary to consider the information provided in the navigation message to determine the accurate travel time [2]. The exact structure and contents of the message depend on the GNSS system and are not necessary knowledge in the scope of this work and thus are not considered here.

2.3 GNSS Receiver

A receiver in GNSS systems is responsible for receiving the signals transmitted by the satellites. These signals are processed in the receiver and the information within the signals is then extracted and used for positioning. Since the actual receiver is rather complex and this thesis focuses on the relatively small correlator part, the receiver is considered on a very general level.

The following figure 2.4 presents the basic structure of a GNSS receiver. The first part along the signal path of the receiver is the antenna and possible accompanying electronics such as a low-noise amplifier. After the signal is received by the antenna, it is in analogue form and enters the front-end part of the receiver which is responsible for operations such as filtering, amplification and down-conversion of the signal to a lower frequency. In the end of this stage the signal is digitized by an analog-to-digital converter for further processing [1].

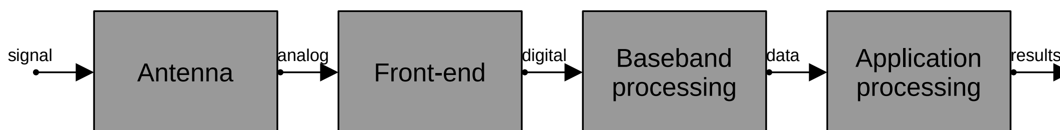

Figure 2.4. *Basic structure of a GNSS receiver*

After this, the signal is in complex base-band form and the acquisition and tracking of the signals take place. This is done in the baseband processing block. This allows the system to determine which satellite signals can be found in the received signal, as well as to determine the transmit time of the signals and to extract the navigation messages. Finally, all the gathered information is combined and processed to get meaningful positioning results [1].

2.4 Signal acquisition using correlation

Signal acquisition is performed by taking advantage of the constantly repeating ranging code in the signal that is unique to each satellite. Since these codes are known, it's possible to calculate the correlation between the code and the captured signal to determine which satellite signals are present.

During signal acquisition, the sampled signal is correlated with a known reference code, but since the phase of the signal is not known, it's necessary to test different reference code phases to find the correct one. Correlation is then calculated over these different phase offsets and once the correlation result exceeds a certain threshold value, the signal and the correct phase is considered to be found. The search for the correct phase offset is done using different phase values. For example, spacing of half a chip can be used in this process. As different code phases are searched, it's also necessary to adjust the frequency [2]. This is done due to the Doppler shift of the frequency, which is caused by the velocity of the satellite and the receiver towards or from each other [3]. So overall, the correlation must be calculated for different code phase and Doppler shift values. The Doppler shift and code phase value pairs are also known as cells. In the acquisition process it's necessary to search over a range of different cells. Once the correlation peak is found, so is the signal. In general form, a circular cross-correlation of two discrete signals can be expressed as follows.

$$z(k) = \sum_{m=0}^{N-1} \overline{x(m)} \cdot y[(m+k)_{\mathrm{mod}N}] \tag{2.1}$$

In equation 2.1 $\overline{x(m)}$ and $y[(m+k)_{\text{mod}N}]$ are time-domain signals that are correlated and $z(k)$ is the correlation result. In the scope of GNSS, the x and y are the received signal and the locally generated replica of the code sequence. The bar over the first signal denotes a complex conjugation and in the case where $\overline{x(m)}$ is real valued, $\overline{x(m)} = x(m)$. From the equation, it can be seen that for a correlation of length N, the number of required multiplications followed by additions is N. A complete correlation search over $k = 0, ..., N-1$ would require N^2 multiplications and additions. This number increases rapidly as the length of the sequences grows.

The equation 2.1 presents a circular cross-correlation which assumes that the correlated signals are periodic [4]. While the PRN sequences are periodic, this is not the case for the complete GNSS signals due to the navigation message. To calculate correlation for non-periodic sequences, linear cross-correlation, which is shown below, is used.

$$z(k) = \sum_{m=0}^{N-1} \overline{x(m)} \cdot y(m+k) \tag{2.2}$$

Now, the difference between the linear cross-correlation in equation 2.2 and circular cross correlation is that in equation 2.2, the signals are not treated as periodic. For this reason, the index of the sequence y lacks a modulo operator and thus the indexing doesn't loop back to beginning.

3 FAST FOURIER TRANSFORM BASED CORRELATION

Traditionally the correlation in the signal acquisition phase has been calculated in time-domain. The goal in this thesis is to explore the possibility to calculate the correlation in frequency-domain using Fast Fourier Transform. For this purpose, it's necessary to first understand how the correlation is calculated in frequency-domain, how the FFT fits into this, and what kind of possibilities it has.

3.1 Frequency-domain correlation

According to the convolution theorem [4], Fourier transform of circular convolution of two periodic sequences is equivalent to multiplication of the Fourier transforms of those sequences. In a similar manner, Fourier transform of circular cross-correlation of two sequences is equivalent to multiplying the Fourier transform of the first sequence with the complex conjugate of Fourier transform of the other one [4]. This is shown in the following equation 3.1

$$\mathcal{F}\{f \star g\} = \mathcal{F}\{f\} \cdot \overline{\mathcal{F}\{g\}}, \tag{3.1}$$

where f and g are the functions that are correlated, $\mathcal{F}$ indicates a Fourier transform and $\star$ denotes correlation. $\overline{\mathcal{F}\{g\}}$ means a complex conjugate of the Fourier transform.

Since the equation 3.1 results in a circular cross-correlation, it assumes that the sequences are periodic. In order to get a linear correlation using Discrete Fourier Transform (DFT) and Inverse Discrete Fourier Transform (IDFT), it is first necessary to extend the sequences to a length of $N \geq M + L - 1$, where M and L are the individual lengths of the sequences. While this is still a circular correlation, for one period the result is the same as for linear correlation. [4]. The correlation for two discrete sequences can be computed through the following steps.

1. Zero-pad the sequences $x(n)$ and $y(n)$.
2. Apply DFT to both sequences to get $X(k)$ and $Y(k)$.
3. Multiply $X(k)$ with the complex conjugate of $Y(k)$.
4. Apply IDFT to the multiplication result to get the non-circular correlation $z_{corr}(n)$.

In the previous steps, lower case letters indicate time-domain and upper case indicates frequency-domain. The IDFT operation gives the correlation results, but in GNSS search engine it's useful to take a magnitude of the correlation result. This magnitude operation is included in the proposed design and is shown in the picture below.

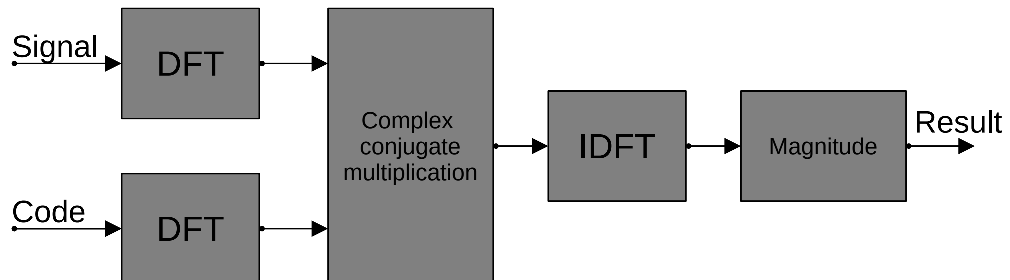

Figure 3.1. *Frequency-domain correlation in GNSS system*

Figure 3.1 visualizes the stages that are required for the correlation process. This gives an idea what are the stages and major elements of the frequency-domain correlator unit.

3.2 FFT and Cooley-Tukey algorithm

Calculation of a DFT can be sped up by using FFT algorithm. To understand how this is done, first consider a complex Fourier series which be calculated with the following equation.

$$X(k) = \sum_{n=0}^{N-1} x(n) \cdot W_N^{kn}, \quad k = 0, 1, ..., N-1, \tag{3.2}$$

where $x(n)$ is the sequence in time-domain with length N, $X(k)$ is the Fourier transform of that sequence and $W_N = e^{-2\pi i/N}$ is the so-called twiddle factor. Correlation search with the equation 3.2 requires N^2 complex operations, meaning multiplication followed by addition. If $N = r1 \cdot r2$, where $r1$ and $r2$ are integer factors of N, the Fourier series can be decomposed into smaller DFTs and this can be continued until all factors are prime numbers. Now taking advantage of the symmetries with the coefficients W, it is possible to lower the computational complexity down to $NlogN$ operations. For example, if N is a power of two, with Radix-2 algorithm the transform requires $Nlog_2N$ operations [5]. As an example, let's have a look at the decomposition of an N-point DFT, $(N = 2^l, l = integer)$,

into two $N/2$-point DFTs.

$$\begin{aligned} X(k) &= \sum_{n=0}^{N-1} x(n) \cdot W_N^{kn}, \quad k = 0, 1, ..., N-1, \\ &= \sum_{n_{even}} x(n) \cdot W_N^{kn} + \sum_{n_{odd}} x(n) \cdot W_N^{kn} \\ &= \sum_{r=0}^{N/2-1} x(2r) \cdot W_N^{2kr} + \sum_{r=0}^{N/2-1} x(2r+1) \cdot W_N^{k(2r+1)} \\ &= \sum_{r=0}^{N/2-1} x(2r) \cdot W_N^{2kr} + W_N^k \cdot \sum_{r=0}^{N/2-1} x(2r+1) \cdot W_N^{2kr} \end{aligned} \tag{3.3a}$$

Since

$$W_N^2 = e^{\frac{-i2(2\pi)}{N}} = e^{\frac{-i2\pi}{N/2}} = W_{N/2}, \tag{3.3b}$$

it's possible to write the equation 3.3a as

$$\begin{aligned} X(k) = &\sum_{r=0}^{N/2-1} x(2r) \cdot W_{N/2}^{kr} + W_N^k \cdot \sum_{r=0}^{N/2-1} x(2r+1) \cdot W_{N/2}^{kr}, \quad k = 0, 1, ..., N/2-1 \\ &= G(k) + W_N^k H(k) \end{aligned} \tag{3.3c}$$

Now the N-point DFT $X(k)$ is expressed with $N/2$-point DFTs $G(k)$ and $H(k)$, which have periodicity of $N/2$. This means that $G(k) = G(k+N/2)$ and $H(k) = H(k+N/2)$. Also since $W_N^{N/2} = -1$ and thus $W_N^{k+N/2} = W_N^k W_N^{N/2} = -W_N^k$, the equation 3.3a, when $k >= N/2$, can now be expressed as

$$X(k+N/2) = G(k) - W_N^k H(k), \quad k = 0, 1, ..., N/2-1. \tag{3.3d}$$

Based on 3.3c and 3.3d, in order to get $X(k)$ and $X(k+N/2)$ from $G(k)$ and $H(k)$, only one multiplication and two additions are needed instead of two multiplications and two additions [4].

The decomposition process of the Fourier series is visualized in the figure 3.2.

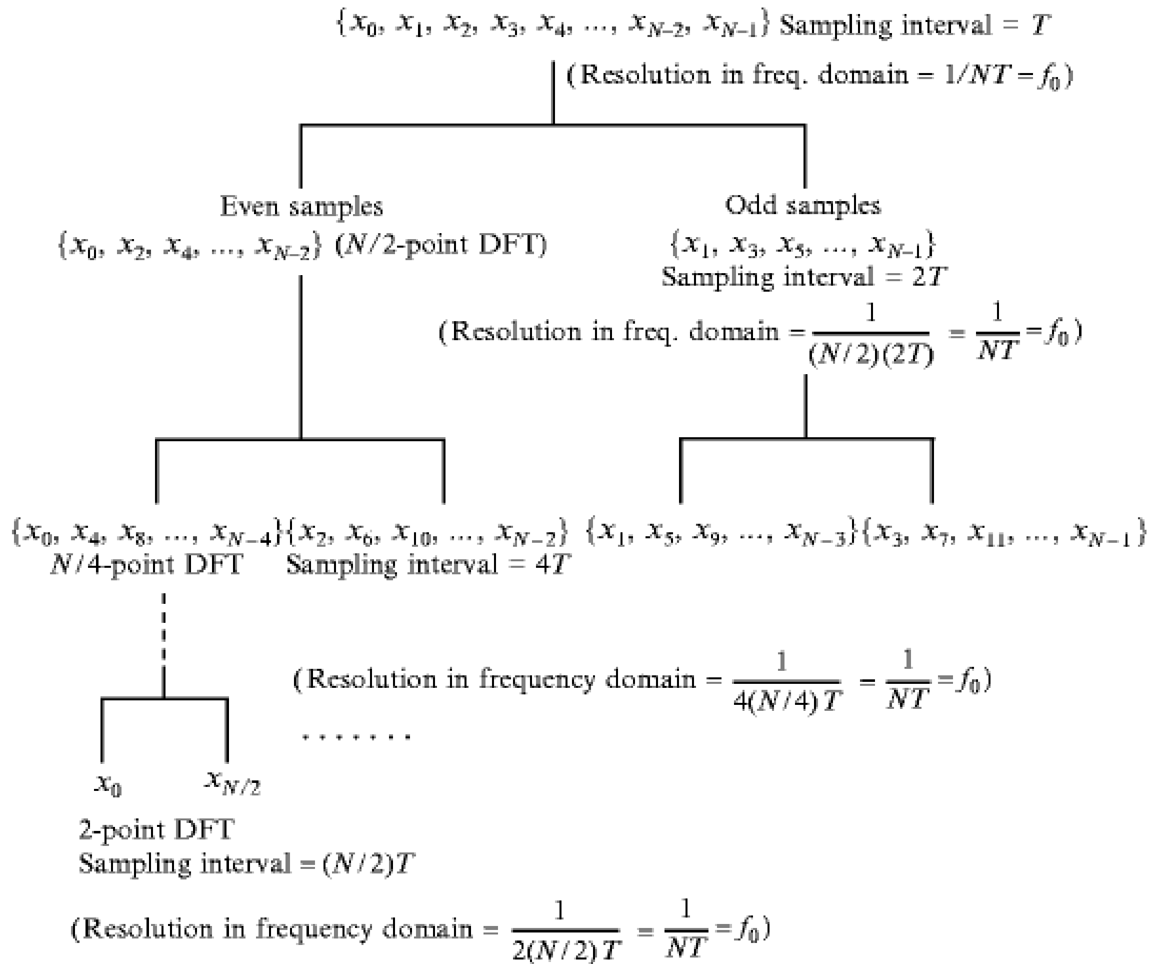

Figure 3.2. *Decomposition of N-point DFT [4]*

For an N-point DFT, where $N = 2^l, l = integer$, the process begins by decomposing the N-point sequence into two $N/2$-point sequences where the first one consists of the samples of the N-point sequence with even indices and the second one has the samples with odd indices in the original sequence. Next step is to divide both of the $N/2$-point sequences to into two, which results in four $N/4$-point sequences. This decomposition can be further continued until there are only 2-point DFTs. This type of decomposition is called decimation-in-time (DIT) algorithm [4].

Another option would be to divide the sequences to first and second halves which would result in a slightly different looking function. This would be called a radix-2 decimation-in-frequency (DIF) algorithm [4]. This process is shown below.

$$\begin{aligned} X(k) &= \sum_{n=0}^{N-1} x(n) \cdot W_N^{kn}, \quad k = 0, 1, ..., N-1, \\ &= \sum_{n=0}^{(N/2)-1} x(n) \cdot W_N^{kn} + \sum_{n=N/2}^{N-1} x(n) \cdot W_N^{kn} \end{aligned} \tag{3.4a}$$

With $n = m + N/2$ the second summation can be presented as

$$\sum_{m=0}^{(N/2)-1} x(m+N/2) \cdot W_N^{k(m+N/2)} = \sum_{n=0}^{(N/2)-1} x(n+N/2) \cdot W_N^{k(N/2)} W_N^{kn}. \tag{3.4b}$$

Now 3.4a becomes

$$\begin{aligned} X(k) &= \sum_{n=0}^{(N/2)-1} x(n) \cdot W_N^{kn} + W_N^{k(N/2)} \sum_{n=0}^{(N/2)-1} x(n+N/2) \cdot W_N^{kn} \\ &= \sum_{n=0}^{(N/2)-1} [x(n) + (-1)^k x(n+N/2)] \cdot W_N^{kn} \end{aligned} \tag{3.4c}$$

since $W_N^{k(N/2)} = -1$. For even ($2r$) and odd ($2r+1$) values of k, the equation becomes

$$\begin{aligned} X(2r) &= \sum_{n=0}^{(N/2)-1} [x(n) + x(n+N/2)] \cdot W_N^{2nr} \\ X(2r+1) &= \sum_{n=0}^{(N/2)-1} [x(n) - x(n+N/2)] \cdot W_N^{n} W_N^{2nr}. \end{aligned} \tag{3.4d}$$

Because $W_N^{2nr} = e^{\frac{-j2\pi 2nr}{N}} = e^{\frac{-j2\pi nr}{N/2}} = W_{N/2}^{nr}$,

$$\begin{aligned} X(2r) &= \sum_{n=0}^{(N/2)-1} [x(n) + x(n+N/2)] \cdot W_{N/2}^{nr} \\ X(2r+1) &= \sum_{n=0}^{(N/2)-1} [x(n) - x(n+N/2)] \cdot W_N^{n} W_{N/2}^{nr}. \end{aligned} \tag{3.4e}$$

Which means that $X(2r)$ is $\frac{N}{2}$ - point DFT of $[x(n) + x(n+N/2)]$, where $r, n = 0, 1, ..., \frac{N}{2} - 1$. In a same way $X(2r+1)$ is $\frac{N}{2}$ - point DFT of $[x(n) - x(n+N/2)]W_N^n$, where $r, n = 0, 1, ..., \frac{N}{2} - 1$.

Now the N - point DFT can be expressed as two $N/2$ - point DFTs. This decomposition results in similar reduction in arithmetic operations as the DIT algorithm [4].

3.3 Inverse transform

Previously presented decomposition can also be used for the inverse transform since the only differences between DFT and IDFT are the sign of the imaginary part of the twiddle

factor and scaling.

$$x(n) = \frac{1}{N} \sum_{k=0}^{N-1} X(k) \cdot W_N^{-kn}, \quad n = 0, 1, ..., N-1, \tag{3.5}$$

Equation 3.5 shows an IDFT, where $x(n)$ is a sequence in time-domain, $X(k)$ is its Fourier transform and $\frac{1}{N}$ is a normalization factor. N is the length of the sequence and W_N is once again the twiddle factor as in equation 3.2. The important difference is the minus sign in the twiddle factor exponent.

3.4 Radix and butterfly unit

In the previous chapter a DFT is decomposed by dividing the sequence into two smaller ones. If this is operation repeated, it results in 2-point DTFs. This is called a radix-2 FFT algorithm. The decomposition process yields an equation that contains two additions and one multiplication. This can be visualized as a data flow graph called a butterfly diagram due to it's shape. The butterfly diagram of a DIT algorithm can be seen in figure 3.3.

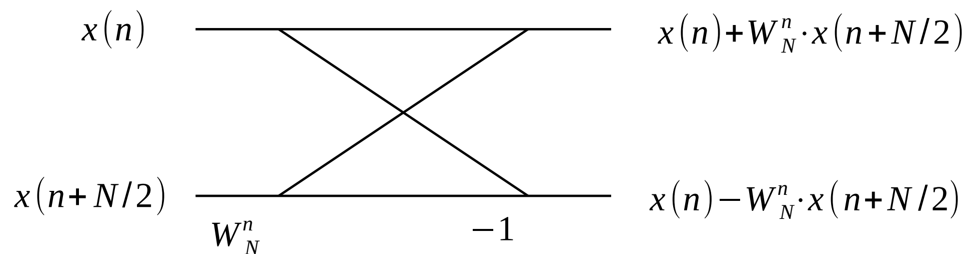

Figure 3.3. *Radix-2 DIT butterfly unit*

In the diagram the inputs are on the left side and the outputs are on the right. The diagonal lines represent addition and W_N^n and -1 are multipliers for the lower input. A butterfly diagram for a DIF algorithm can be seen below in figure 3.4 [4].

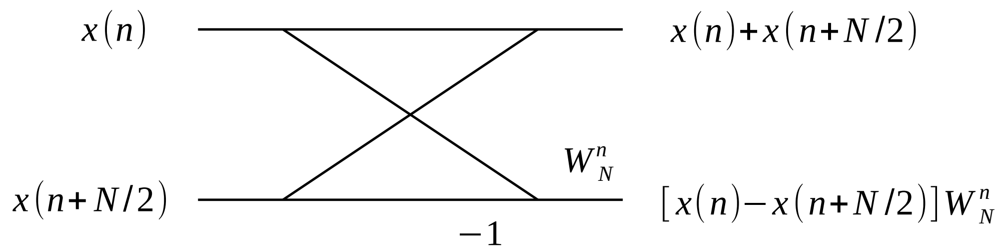

Figure 3.4. *Radix-2 DIF butterfly unit*

The DIF butterfly has the same shape as the DIT butterfly, but differs in the location of the twiddle factor multiplication. In DIF algorithm it takes place after the addition operation.

It is also possible to decompose the DFT by using different radix than two. When N is for example a power of three, it's possible to use radix-3 algorithm. Same goes for other radices such as four or five. The principle remains same, but the resulting butterfly unit takes a more complex form than the rather simple radix-2 version. As an example, in a radix-4 DIF - algorithm, the DFT is divided into four smaller DFTs and the resulting decomposition can be written as below [4].

$$\begin{aligned}
X(4k) &= \sum_{n=0}^{\frac{N}{4}-1}[x(n) + x(n+\frac{N}{4}) + x(n+\frac{N}{2}) + x(n+\frac{3N}{4})]W_{N/4}^{nk}, \\
X(4k+1) &= \sum_{n=0}^{\frac{N}{4}-1}([x(n) - jx(n+\frac{N}{4}) - x(n+\frac{N}{2}) + jx(n+\frac{3N}{4})]W_N^{n})W_{N/4}^{nk}, \\
X(4k+2) &= \sum_{n=0}^{\frac{N}{4}-1}([x(n) - x(n+\frac{N}{4}) + x(n+\frac{N}{2}) - x(n+\frac{3N}{4})]W_N^{2n})W_{N/4}^{nk}, \\
X(4k+3) &= \sum_{n=0}^{\frac{N}{4}-1}([x(n) + jx(n+\frac{N}{4}) - x(n+\frac{N}{2}) - jx(n+\frac{3N}{4})]W_N^{3n})W_{N/4}^{nk}, \\
& k = 0, 1, ..., \frac{N}{4} - 1
\end{aligned} \tag{3.6}$$

Based on the equations 3.6, it's possible to draw a butterfly diagram for radix-4. This is shown in figure 3.5.

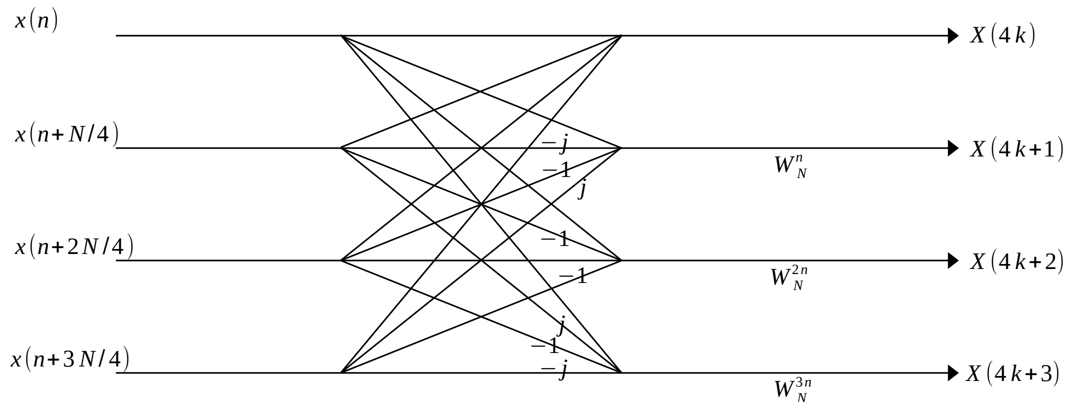

Figure 3.5. *Radix-4 DIF butterfly unit*

This butterfly diagram can alternatively be represented using radix-2 like structures as can be seen below.

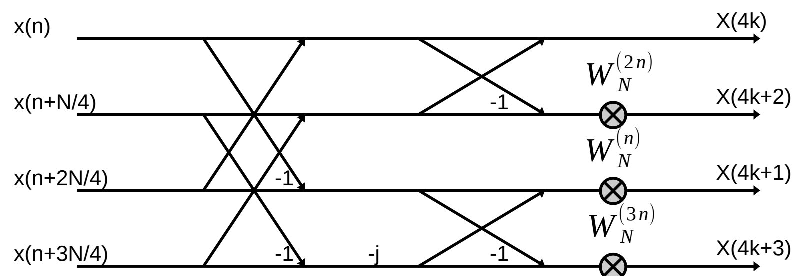

Figure 3.6. *Alternative version of a radix-4 DIF butterfly unit*

In figure 3.6 the butterfly operation now includes three complex multiplications and 12 complex additions. An added benefit of using a higher radix is that the number of stages required for the transform is smaller than compared to a lower radix for the same length N. Also the total number of butterfly operations is lower. These allow potential speed-up of the transform computation.

3.5 Zero padding and mixed radix

Since the previous algorithms require N to be a power the radix used, it is necessary to pad the sequence with zeroes if the length doesn't fulfill this requirement. This has the downside of increasing the length of the sequence and thus the storage requirements for the data. In the worst case this length increase can be very large. Another option would be to use different radices for different stages of the decomposition to support lengths that are composite of these radices. This is called a mixed-radix algorithm and an example of this can be seen in the following figure.

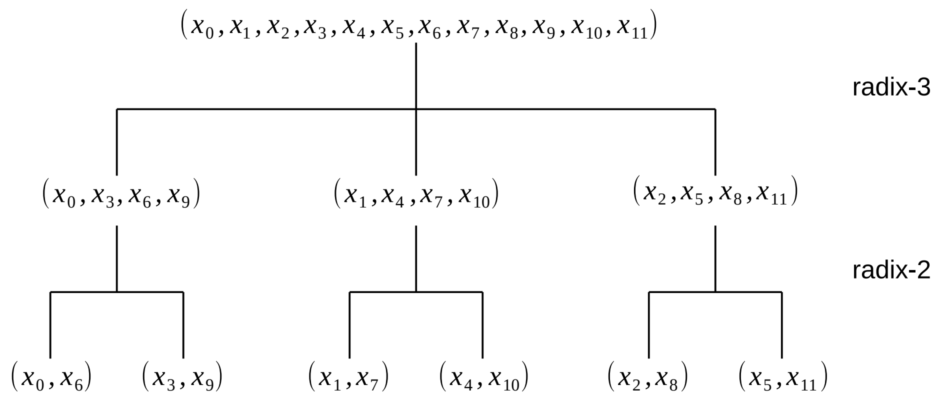

Figure 3.7. *Radix-2,3 DIT decomposition*

Figure 3.7 shows the decomposition of 12-point DFT using radices 2 and 3. First the sequence is decomposed by a factor of three and after that with a factor of two. If just a radix-2 algorithm was used, it would be necessary to first pad the input sequence to a length of 16. Thus, mixed-radix algorithm offers improvements by reducing the number of butterfly calculations as well as lowering the memory requirements since shorter transform length is possible.

3.6 In-place computation

In-place computation means that the algorithm uses only one data-array and the output is stored within the same array in the same locations where the inputs are located, overwriting the previous values. Another option would be to store the output in different data structure, but this increases the memory requirements. When calculating FFT using an in-place algorithm, the order of the data sequence changes. DIT algorithm requires the input to be in bit-reversed order and produces the result in natural order, while for DIF, the input is in natural order and the output sequence is in bit-reversed order [4].

An example of a radix-2 FFT algorithm is presented in program 3.1. A simple way to implement control for FFT is with three nested loops. The topmost loop goes through the stages of the decomposition, second one loops through "segments" and finally the innermost loop iterates through each butterfly operation in each segment, as seen below.

```
1 function X = fft_rad2nested(x)
2     np = nextpow2(numel(x)); % next power of 2 for zero padding
3     x = [x zeros(1, 2^np - numel(x))]; % pad with zeros
4     N = 2^np; % Length of signal
5     stages = np; % number of stages
6
```

```
7       x = bitrevorder(x); % Reorder data
8       for stage = 1:stages
9           items = 2^stage; % items per segment
10          segments = N / items; % segments per stage
11          for segment = 0:segments-1
12              btrflies = items / 2; % butterflies per segment
13              for btrfly = 0:btrflies-1 % loop trough butterflies
14                  % Twiddle factor
15                  twiddle = exp(-2i*pi*btrfly / items);
16                  % calculate indexes
17                  idx1 = btrfly + segment*items + 1;
18                  idx2 = btrfly + segment*items + items/2 + 1;
19                  % read inputs
20                  x1 = x(idx1); % first input to the butterfly
21                  x2 = x(idx2); % second input to the butterfly
22                  % butterfly operations and store results
23                  TWx2 = twiddle * x2; % multiply with twiddle
24                  x(idx1) = x1 + TWx2;
25                  x(idx2) = x1 - TWx2;
26              end
27          end
28      end
29      X = x; % Return
30  end
```

Program 3.1. *Matlab code for FFT with three nested loops*

In the beginning of the code, input signal is padded with zeroes to a power of two length. After that, on line 5, the necessary number of stages is decided. As the code uses a DIT algorithm, it's necessary to reorder the input which is done on row 7. Rest of the code includes the three loops for stages, segments and butterflies as well as the computation of support variables for these loops. The actual butterfly operation takes place inside the innermost loop on rows 23-25.

As mentioned, the code implements a radix-2 FFT computation, but it could be easily modified to also do an IFFT calculation since the only difference between the transforms is in the twiddle factor and scaling. By taking a complex conjugate of the twiddle factor calculated on row 15, an IFFT could be computed instead.

3.7 Hardware FFT

For the hardware implementation of the FFT/IFFT, there are two basic options to choose from. These are memory-based in-place design and a pipelined design which are discussed in the following sections.

3.7.1 Pipelined FFT

In a pipelined FFT, the computation of the different stages can take place simultaneously. All the stages are implemented as separate processing elements and the data moves as a stream into the first stage which outputs it to the next stage and so on until finally the results are streamed out of the last stage output. One example of a pipelined architecture is a single-delay-feedback FFT which can be seen below in figure 3.8.

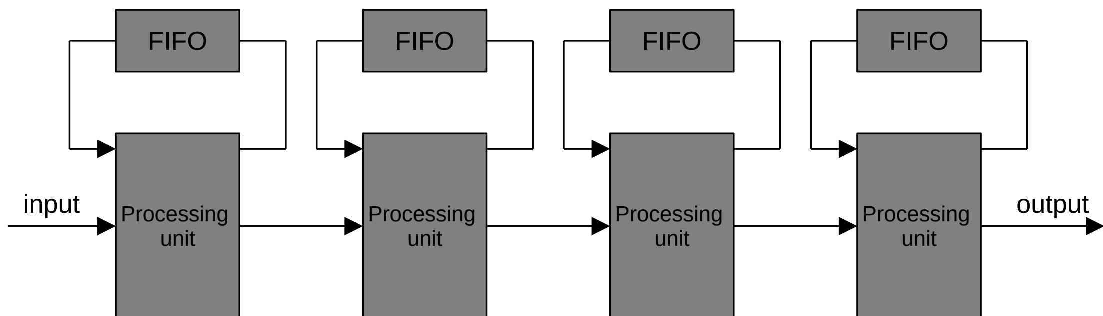

Figure 3.8. *Single-delay-feedback FFT*

In this design, each of the stages consists of a processing unit and a delay loop. The processing unit contains a butterfly unit and other necessary logic for control and the twiddle factors. Using the feedback, some of the incoming samples are delayed and sent back to the other input of the processing unit where they arrive at the same time as the corresponding input arrives. At this point the butterfly calculation finally takes place. Of the two results, the first one is forwarded to the next butterfly while the second one is put into delay-line before it's eventually also forwarded to the next butterfly [6]. The maximum FFT length is decided by the radix and the number of stages in the pipeline structures. Shorter FFTs could be computed by skipping stages in the beginning or in the end, depending on wether the design uses DIF or DIT algorithm. In the picture the delay line is implemented with first-in-first-out (FIFO) elements. The size of the FIFO depends on the corresponding stage and is equal to the distance between butterfly input pairs and from the example program 3.1 it can be seen that this is equal to $2^{stage-1}$, where stage is the index of the stage in the FFT. Since each sample has to go through the pipeline and

the FIFOs, once the sequence is loaded in the pipeline, it would take approximately N cycles to get the result.

A benefit of a pipelined FFT is somewhat simple addressing, since it's done by using the delay that is constant for each stage. This accompanied with simultaneous stage calculations allows the architecture to achieve generally higher throughput than what a memory-based design offers. As a downside, this performance comes with a price since the design requires a dedicated butterfly unit and twiddle factor generation for each of the stages. This results in typically higher design area and power consumption. Also the presented example doesn't optimize the butterfly utilization since due to the feedback loop, the butterfly unit in each processing unit is used only during half of the total cycles.

3.7.2 Memory-based FFT

The other option is the memory-based architecture mentioned earlier. With this design the memory accesses are problematic in terms of performance, since both inputs need to be read from the memory and the results written back into the memory. This is time consuming if only one memory operation can be performed simultaneously. This can be improved by using a dual port memory to allow read and write operations at the same time. Another improvement would be to divide the memory into two banks so that one input is read from the first bank and the other input from the second one. In same way the results are written in different banks. This makes it possible to access both values at the same time and reduce number of required clock cycles.

On the other hand, this memory partitioning complicates the addressing logic. For example, if same address generation logic is used as in program 3.1, but now with divided memory, during the first stage the two inputs are read from separate memory blocks. The results are then stored in the same addresses. Problem is that now the system would end up having the butterfly input pair in the same memory block [7], as can seen below.

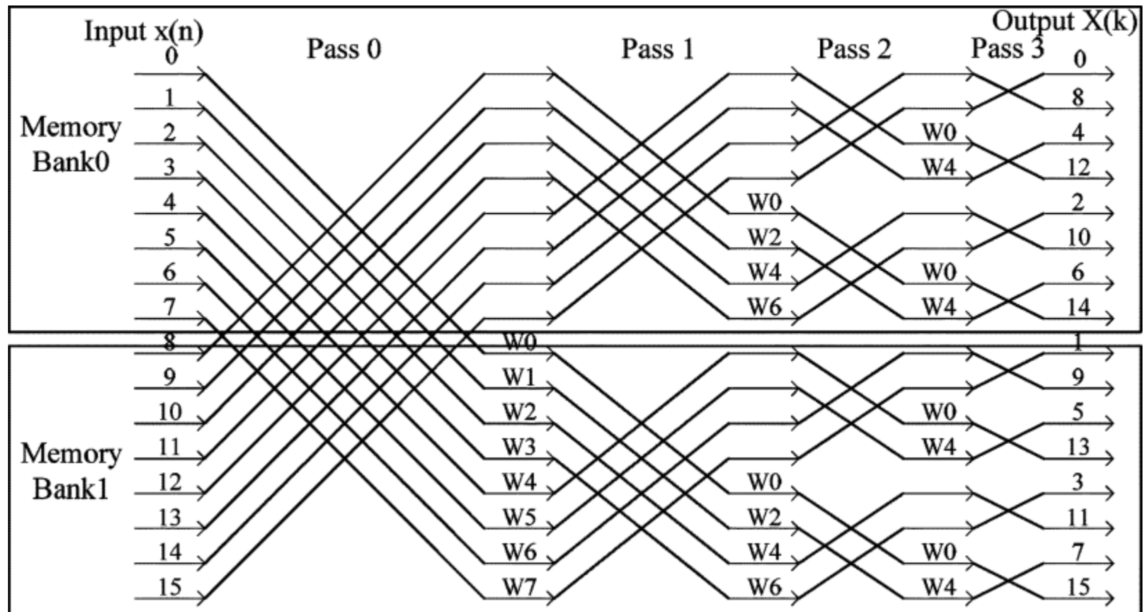

Figure 3.9. *16-point radix-2 DIF FFT [7]*

In figure 3.9 it can be seen that on the left side, the data is originally divided into two memory banks in a way that during the first stage the inputs are in different banks. Butterfly operation is represented as the crossing lines and for example the first butterfly gets the samples zero and eight as its inputs. The outputs are then stored in the same locations. During the second stage, the first butterfly takes samples zero and four, but these are now located in the same memory bank, so simply dividing the memory doesn't solve the memory access problem. It can however be solved with a different kind of address generation logic [7].

Even though the memory-based architecture requires more complicated address generation, it can be implemented using only one butterfly unit. This saves area compared to the pipelined approach, but also makes the FFT computation slower.

4 INTRODUCTION TO HIGH-LEVEL SYNTHESIS

Now that there's an understanding of what is the function of the correlator and how it can be implemented using FFT, it's time to move on to the actual implementation. The design is created using high-level synthesis and in the following sections the high-level synthesis is briefly presented.

4.1 High-level synthesis

In the beginning of a digital hardware design flow, there exists a behavioral description of the design which explains what is the intended function of the system. During the design flow this is followed by architectural decisions that describe how this functionality is implemented. In other words, they focus on the technological aspects of the system such as the performance and area. Finally these choices are implemented with a hardware description language such as Very High Speed Integrated Circuit Hardware Description Language (VHDL) or Verilog. Writing this register transfer level (RTL) description however is a time-consuming task and as the complexity of the systems increases, so does the time required for writing the code. Especially since in the beginning it is not known what is the best way to implement the design, it might be necessary to re-write the code. Growing code length also means higher possibility of bugs which again increases the verification load [8].

HLS aims to solve these problems by automating the generation of RTL directly from a higher level language. A benefit of higher abstraction level is that a lot of details can be left out. For example, it's not necessary to worry about clock signals or technology onto which the design is mapped. This also has the added benefit of shorter code, which reduces the number of errors and lowers the verification effort. HLS also benefits the re-usability of the code since without technology dependent details, the design can be easily targeted to another technology. Making changes and adding new functionalities to the system is also made easier. As the amount of work to generate RTL is reduced, time can be allocated to other tasks such as algorithm and architecture development and verification [8].

For this thesis, Catapult High-Level Synthesis by Siemens [9], which supports C and C++ as a high-level language, is chosen. Examples for other HLS tools are Vivado High-Level Synthesis by Xilinx [10], Intel HLS compiler [11] and Stratus HLS [12], which all support

C++ language. One example for a tool with another HLS language would be HDL Coder by MathWorks which generates RTL from MATLAB code [13].

4.2 Catapult HLS

One upside of using high level synthesis is the ability to map the design to multiple platforms as well as to use different goal clock frequencies. By using various different technology libraries, it is possible to generate the RTL for field-programmable gate arrays (FPGAs) as well as for application-specific integrated circuit (ASIC) technologies. Catapult comes with premade libraries for FPGAs from multiple vendors as well as some ASIC technology libraries. It is also possible to create additional ones for ASIC technologies and memory architectures.

During the RTL generation, the desired technology and memory libraries are first chosen. This is followed by configurations for architectural decisions, such as clock speed, interface types and memory resources and structures. Performance can also be affected by choosing how the loops in C++ code are pipelined and how much parallelism is wanted. Catapult makes architecture exploration easier by showing how timing information about the design and how the operations are schedule as well as an estimate of the resource usage.

It is possible to launch other tools needed for the design flow directly from Catapult. Catapult supports various different synthesis and simulation software. For simulating the design, Xcelium Logic Simulation by Cadence is used. When launched from Catapult, the simulation results are compared to the ones from running the C++ code. For synthesizing the design for ASIC platform, Genus Synthesis Solution by Cadence [14] software is used.

In this thesis, the design is generated for the Stratix V FPGA board and finally also for a 28nm ASIC technology.

5 CORRELATOR USING HLS

In the following section the process of creating a correlator design using high-level synthesis is described. Also several design aspects and problems that were encountered along the process are presented.

5.1 From algorithmic model to hardware

In the beginning, an algorithmic model of the correlator is created. As seen in chapter 3.1, the FFT based correlation looks rather straightforward. Based on the figure 3.1, the correlator design can be divided into three sub-blocks: the block that calculates the FFT and IFFT, multiplication block, and finally the block that takes the magnitude of the result. The general principle of the correlator can be easily tested with built-in functions of Matlab. The most complex part of the design lies in the FFT/IFFT block. The next step in the process is to choose a design for the FFT block.

As mentioned in chapter 3.7, the options for the FFT implementations are memory-based and a pipelined architecture. Since compromises had to be made between speed and area, I ended up choosing the memory-based architecture for my implementation of the FFT since it results in smaller area than the pipelined version.

If the previously presented program 3.1 is considered, it looks quite simple, but this algorithm has one major downside. As it reads the inputs from and stores the outputs in one single array, it implements an in-place algorithm. If created as hardware, in order to calculate one single butterfly operation, it would need to spend two cycles for reading the two input values from the different memory addresses and two more to store the results back in the same memory addresses if single port memory is used. This makes the design too inefficient and thus it's necessary to come up with something better.

5.1.1 FFT address generation

As mentioned, the in-place algorithm has some challenges. Over the years several different addressing schemes have been developed that try to overcome these issues. For this design, the address generation logic presented in [7] is chosen due to relatively simple design and small resource usage.

In [7], the issue with the memory banks is solved by adding exchange circuits before and after the butterfly unit. When the control signal to the exchange circuit is one, the data is swapped between memory banks. Otherwise, locations of the data remain the same. Controls are chosen in such a way that the data inputs and outputs for a given butterfly are always in different memory banks. Signal flow graph with this addressing logic is visible below.

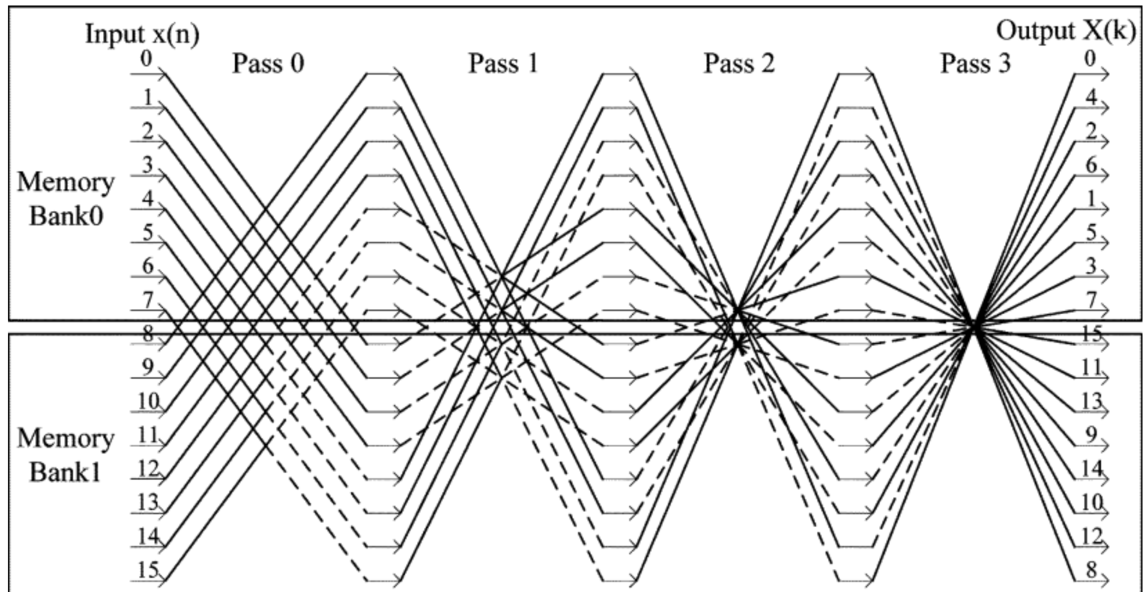

Figure 5.1. *16-point radix-2 FFT with exchange circuits [7]*

For every butterfly in figure 5.1, the inputs and outputs are located in different memory banks. Dashed lines in the figure represent exchange of data and the solid lines indicate that the data locations remain unchanged.

In this design, the read addresses are same as the write addresses. For the memory bank $M0$, the address is B, which is the value of the butterfly counter that counts from zero to $N/2 - 1$. The address for the bank $M1$ is computed by inverting P amount of bits of the B value, starting from the most significant bit. P is the so-called pass counter that counts from zero to $log2(N) - 1$, and its value is the current stage. For example, during the stage 0, the addresses for banks $M0$ and $M1$ are the same. During stage 1, address for $M1$ is computed by inverting the MSB of the $M0$ address. For stage 2, to get $M1$ address, it's necessary to invert two most significant bits and so on for the rest of the stages. During the first stage the input exchange control signal $C1$ is always zero and during the last stage the output exchange control $C2$ is always zero. Apart from those exceptions, the $C1 = b_{r-s-1}$ and $C2 = b_{b-s-2}$, where b is a single bit in the butterfly counter and its index is computed based on the $r = log2(N)$ and the current stage s. Additionally, if a memory is used for storing the twiddle factors, the address for twiddle factor for a stage s is $b_{r-s-2}b_{r-s-3}...b_0 0...0$, where the s least significant bits are zeros [7].

Example of this address generation can be seen below.

Table 5.1. *16-point FFT address generation.*

		Stage 0, s=0 C1=0, C2=b_2		**Stage 1, s=1** C1=b_2, C2=b_1		**Stage 2, s=2** C1=b_1, C2=b_0		**Stage 3, s=3´** C1=b_0, C2=0	
Counter $B(b_2b_1b0)$	**Counter** $\overline{B(b_2b_1b_0)}$	Bank 0 address $b_2b_1b_0$	Bank 1 address $b_2b_1b_0$	Bank 0 address $b_2b_1b_0$	Bank 1 address $\overline{b_2}b_1b_0$	Bank 0 address $b_2b_1b_0$	Bank 1 address $\overline{b_2b_1}b_0$	Bank 0 address $b_2b_1b_0$	Bank 1 address $\overline{b_2b_1b_0}$
000	111	000	000	000	100	000	110	000	111
001	110	001	001	001	101	001	111	001	110
010	101	010	010	010	110	010	100	010	101
011	100	011	011	011	111	011	101	011	100
100	011	100	100	100	000	100	010	100	011
101	010	101	101	101	001	101	011	101	010
110	001	110	110	110	010	110	000	110	001
111	000	111	111	111	011	111	001	111	000

In the table 5.1, it is visible, how the addresses are generated for a 16-point FFT by inverting the butterfly counter bits and how the exchange control signal values are chosen. For larger transform sizes, the number of bits in the butterfly counter increases, but the logic remains the same.

5.1.2 Bit reversal issue

As mentioned earlier, the FFT algorithm reorders the data. If the input is in natural order, the output would be in bit-reversed order and vice-versa. The correlation requires that FFT operation is followed by multiplication and finally by IFFT. This means that if a DIF algorithm is used for the FFT, the data would be in bit-reversed order at the IFFT input, which wouldn't work for a DIF algorithm. In order to counter this effect, it would be necessary to implement a bit-reversal algorithm to reorder the data. Especially since the FFT design is required to support different transform lengths, this would add additional computation and latency to the system.

Another option, that I ended up using in this thesis, is to use different algorithms for FFT and IFFT. Now if DIF is used for FFT and DIT for IFFT, data reordering isn't necessary anymore since now FFT produces the results in bit-reversed order, which is the required input order for DIT. And since IFFT reorders the data, final correlation results are in natural

order. Because the address generation logic that is used, calculates the data addresses based on the stage, it is only necessary to iterate through the stages in reversed order, starting from the highest one and ending at the stage zero, to get the correct addresses for the DIT algorithm instead of the DIF. The biggest change is to expand the butterfly unit to implement also the DIT computation shown in the figure 3.3. This requires adding control signal to indicate wether the FFT algorithm currently used is DIT or DIF.

5.1.3 C++ code structure

With the algorithm decided, the next stage is to write a Matlab code of the FFT/IFFT to verify that the algorithm works as intended. For easier transfer of the code to C++ language, it's useful to write the code without using too much built-in Matlab functions. Once working algorithmic representation is obtained, the C++ HLS code is written based on the Matlab code.

Before moving on to the code, it's good to mention a special aspect of writing hardware with C++. Just by using regular C++, libraries it would be difficult to describe all the possible aspects of the hardware. Luckily, specific libraries have been created to make this effort easier. While other alternatives, such as SystemC exist, in this thesis Algorithmic C libraries are used. These are a collection of libraries that provide synthesizable implementations of, for example, bit-accurate datatypes, connections between blocks and mathematical functions [15].

The partitioning of the hierarchical blocks in the correlator can be implemented as classes in the C++ code. This means classes are created for the correlator, FFT/IFFT block, butterfly unit and magnitude calculation block. Originally, a separate class was also planned to implement the complex conjugate multiplication needed for of the correlation, but eventually this was integrated inside the FFT/IFFT block. The correlator object acts as the top level of the design and objects for magnitude block and FFT/IFFT block are instantiated inside it. The butterfly object is instantiated inside the FFT/IFFT block. A code example for the top level correlator is shown below.

```
1  class CFftCorrelator{
2
3      private:
4          // Initiate sub-blocks
5          CFftRad248 fftBlock;
6          CMagnitude magnitudeBlock;
7
8          //Interconnects for data
9          // cx Data Unit is a struct that contains complex samples
10         ac_channel<cx_Data_Unit> IFFTResult;
```

```
11
12          //Interconnects for controls
13          // ac int is a bit accurate integer datatype part of the
14          // ac int library
15          ac_channel<ac_int<MAX_BITS_FOR_IDX+1, false> > N1_ch;
16          ac_channel<bool> MagnCtrl_ch;
17
18      public:
19          //Constructor
20          CFftCorrelator(){}
21
22          //Correlator
23          // pragma hls design interface is used for the only public
24          // function of the class, function parameters define the block
25          // inputs and outputs
26          # pragma hls_design interface
27          void CCS_BLOCK(run) //CCS BLOCK() indicates a hierarchical block
28              (cx_Struct &CodeMem,
29              cx_Struct &SignalMem,
30              cx_Struct &OutputMem,
31              ac_channel<control_Struct> &Control_ch)
32          {
33              #ifndef __SYNTHESIS__
34              while(Control_ch.available(1) or
35                  fftBlock.fft_state() == true)
36              #endif
37              {
38                  // FFT/IFFT block
39                  fftBlock.run(Control_ch, CodeMem, SignalMem,
40                                  IFFTResults, N1_ch, MagnCtrl_ch);
41                  // Magnitude block
42                  magnitudeBlock.run(N1_ch, MagnCtrl_ch,
43                                          IFFTResult, OutputMem);
44              }
45          }
46  }
```

***Program 5.1.** C++ code for the correlator class*

The program 5.1 shows the top level correlator class. The sub-blocks for the FFT (fftBlock) and magnitude block (magnitudeBlock) are instantiated inside the private part. Also, in-

terconnections to connect these two sub-blocks together are created using ac_channel types. The IFFTResults channel is responsible for transmitting the results from FFT block to the magnitude block while the N1_ch and MagnCtrl_ch transfer the info about the FFT length and whether or not it's necessary to compute the magnitude. The ac_channel class is provided in the algoritmic C libraries and it implements a FIFO-like object in the code [15]. The actual hardware implementation of this channel can be chosen in the Catapult tool. The channel N1_ch is defined for a ac_int datatype. Inside the second angle brackets is defined the bit width of the datatype, which in this case is MAX_BITS_FOR_IDX + 1. The second parameter "false" defines that it's an unsigned value.

The design works in a way that it reads data straight from an external memory and writes the correlation results into another external memory. The intermediate FFT results are stored internally. In the C++ code these external connections are done by giving the correlator function structs as parameters. These structs hold arrays that represent the external memories. Also, a channel for the control signals can be seen in the function arguments. Finally, inside the run function the sub-blocks are called. The connections between these blocks and the interface of the correlator block can be seen in the function arguments. The run function calls the FFT and magnitude sub-blocks as well as connects them together.

Another interesting detail in program 5.1 is on the rows 33-36. This defines a while loop that keeps the function running until the correlation is done. This part of the code is only used when running the code. The row 33 excludes it from the synthesis and no hardware is generated for it. These kind of structures are sometimes necessary since while the hardware processes are parallel, the code execution is sequential and it wouldn't otherwise necessarily work as intended.

5.1.4 Bit-accurate data types and other HLS optimizations

Another necessary change when rewriting the code from purely algorithmic to HLS, is to use bit accurate data types. These are offered by *ac_int, ac_fixed* and *ac_complex* libraries which are part of Algorithmic C libraries [15]. In order to minimize the area of design, it's preferable to use as short word length as possible. On the other hand, due to the arithmetic operations in the Fourier transform, the bit width would increase at every stage. To avoid possible overflow, the butterfly computation results need to be scaled down so that they fit in the original number of bits. In this design, scaling is done by computationally light bit-shift operation. This is done at every butterfly operation during the FFT and in total it leads to a scaling by the length of the transform. The downside of this is reduced accuracy due to limited number of bits reserved for the fractional part of the number.

Optimization for writing the code for HLS also include creating loops with fixed amount of

iterations, instead of setting the number with a variable. This allows some optimizations, since the HLS software can know the maximum size of the loop. If variable sized loop is needed, it's possible to use conditional break to end the loop early. Another improvement to the design is to use shift operations instead of multiplications or divisions whenever possible [8]. *Ac_math* library [15] also includes HLS optimized alternatives to some math functions, such as division.

5.1.5 Multiply and magnitude blocks

In the original design the function of the multiply block is used to multiply signal FFT samples with the complex conjugates of the code FFT samples. This block also contains a memory to store the FFT results of the code sequence, so there's no need to recalculate it for every single correlation. The block receives samples from the FFT/IFFT block. In the case of code FFT, it stores them in the internal memory and in case of signal FFT, it multiplies the samples with values from the internal memory. Results are sent back to the FFT/IFFT block for the IFFT computation. Data and control signals are transferred using ac_channel object and complex conjugate is taken with the help of built-in member function of the ac_complex datatype class that is used to represent the complex data samples.

In the final design however, the functionality of this block is included in the FFT/IFFT block in favor to simplify the design. Otherwise the FFT/IFFT block would need to read the multiplication results from the channel into it's internal memory while it is still doing the FFT operation. Now, during the last stage, the butterfly outputs are simply multiplied before being written into the internal memory.

The other hierarchical sub-block under the correlator level besides the FFT/IFFT is the block that calculates the magnitude of the IFFT result. While it's not part of the correlation process, it can be useful in the GNSS acquisition process. The block receives two samples at a time, which come directly from the buytterfly output during the last stage of IFFT. Then pre-made mathematical functions provided in ac_math libraries are used for the magnitude process. These include taking the square of the real and imaginary parts of the complex number and finally the square root of sum of those parts [15]. This is computationally heavy and finding a lighter algorithm for the magnitude computation is one possible improvement in the future. Rounding also takes place after the square root operation. The magnitude block receives the control signals from the FFT/IFFT block and it loops through the transform length and stores the results into an external memory.

5.1.6 C++ test bench

One significant benefit of high level synthesis is the ability to use the same test bench for both the C++ code and the generated RTL design. For my design, I wrote a C++ test bench that creates arrays, which the design sees as the external memories for the code and signal input. Also an array for the output memory is created where the correlator writes the correlation results. Additionally, control variables are generated, which indicate the design the length of the input signal and the intended FFT transform length. Another control signal indicates wether or not it's necessary to calculate an FFT for the current code sequence in case one hasn't been previously computed and stored into the internal memory. The PRN codes as well as the test signals are created with a Matlab script and read from a text file into the test bench.

In the original version of the test bench, the arrays were filled with the signal data and then the design was called. After finishing the correlation, results were read from the output array and written to an external file. This was done for multiple different signal and code sequences. Unfortunately this approach caused problems during the simulations since the tool compares the simulation results to the ones gained by simply running the C++ code. For reasons unknown, changing the contents of memory arrays between the loop iterations caused the simulation and C++ run go somehow off-sync. Because of this, the simulated results were compared to C++ results from another loop iteration, which resulted in failed simulation. To avoid this problem, I ended up increasing the size of the signal and code memory arrays to make it possible for them to hold several signal and code sequences simultaneously. Once the test signals are in the arrays, it is possible to compute multiple correlations in a loop and read the results. Since the contents of the input arrays remain unchanged during the whole simulation, the problem with the synchronization doesn't occur. To make this possible, it is also necessary to create control signals that indicates where in the external input memory the current input sequence is located. Using this as an offset for address in the FFT, the design gets correct values during the first stage.

5.2 Final designs

In the previous section 5.1, some aspects of the HLS design are considered and also some of the problems are brought up regarding the design and what kind of algorithms are used. This section takes a look at the final design and it's structure. Also, an additional mixed-radix-248 based design is considered.

5.2.1 Radix-2

The basic structure of the correlator design consists of two sub-blocks. These are the FFT/IFFT - block and the magnitude block that is also responsible of writing the results into an external output memory. Block diagram of the design is presented below.

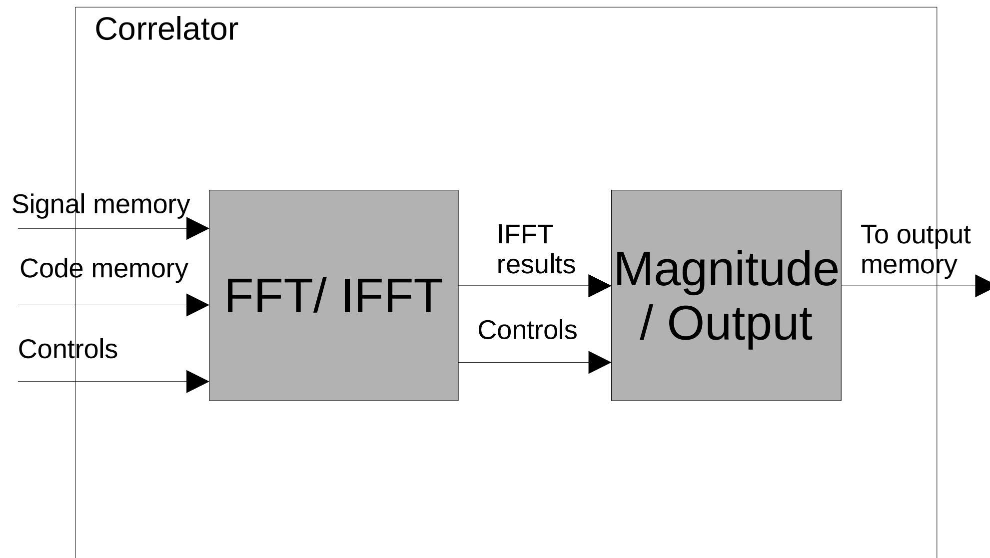

Figure 5.2. *Block diagram of the correlator top level.*

The inputs to the system can be seen from figure 5.2. These include external memories that hold the signal samples as well as the code samples. Other necessary inputs are control signals that tell the FFT block the lengths of the signal and the transform, as well as the location of the input sequence in the external memory. The FFT block also gets information that tells if it's necessary to first calculate the FFT for the code sequence. Of course it would be possible to add internal logic and remove this control signal by adding the ability for the system to decide if it's necessary to do the code FFT or if previously calculated transform exists in the internal memory for the current code. But for simplicity, this is implemented as external control in the current design. A control signal also exists to enable or disable the magnitude calculation.

The external controls first go to the FFT/IFFT block and the controls for the magnitude block are forwarded from the FFT/IFFT block along with the IFFT results. In the output, the system is connected to an external memory where the magnitude block writes the correlation results in natural order.

The design of the magnitude block isn't particularly complex so we'll focus on the transform block which is visualized as follows.

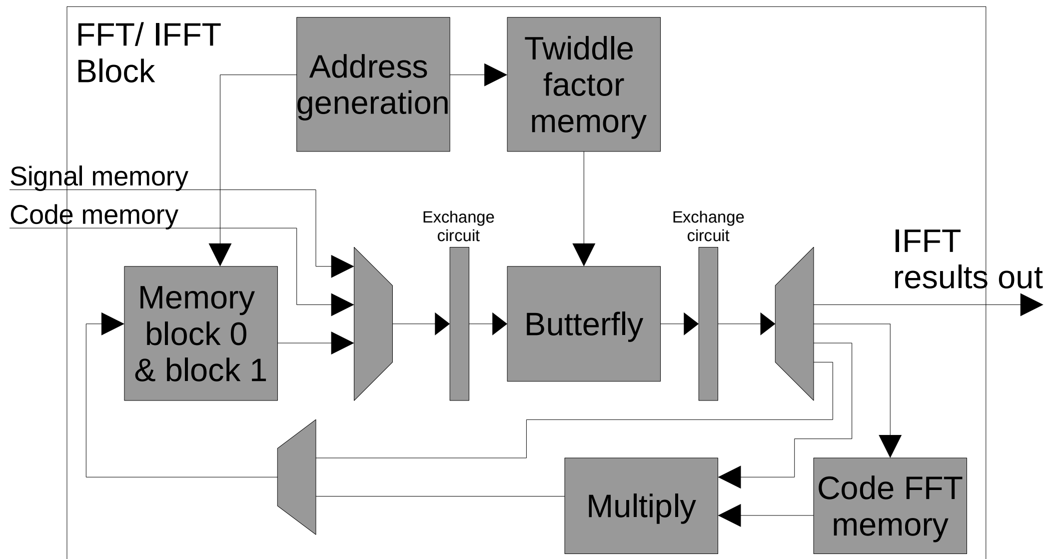

Figure 5.3. *Block diagram of the proposed FFT/IFFT block without control signals.*

In figure 5.3 all the parts are visible that have been mentioned earlier. There are the internal memory, exchange circuits, butterfly unit, address generation and a memory for the twiddle factors.

First, on the left side of the design, there are the connections to external memories. On the inside of the block is the internal memory that holds the intermediate values during the FFT computation. This memory acts both as the input and output memory to the butterfly unit. The internal memory is actually partitioned into two smaller blocks to make it possible to access two values simultaneously. On the top of the block, there is the address generator that is responsible for generating the correct memory addresses for accessing the external and internal memories.

Next part is the logic that decides whether the inputs for the butterfly are taken from internal or external memory. When calculating the FFT, during the first stage the inputs are fetched from the external memory instead of internal. During all the other stages, the inputs are read from the internal memory. This is done to avoid wasting cycles for copying the input signals to the internal memory before starting the FFT computation. On the other hand, if it's assumed that the external memories are not partitioned in a similar manner as the internal one, thus limiting the memory accesses to one per cycle, there is something of a bottleneck which prevents pipelining of the butterfly operations with an interval of one. To minimize the impact of this, it's necessary to separate the first and the last loops from rest of the FFT loop body, so the rest can be pipelined with an interval of one. The stages that are limited by the external memory accesses are then pipelined with higher interval. This way the effects of the bottleneck can be reduced to only two stages.

Next on the data path is the exchange circuit that implements the input swapping. Control signals for the exchange circuits are left out of the diagram for clarity. From there, the data goes to the processing unit that computes the butterfly operation using a twiddle factor that is fetched from the twiddle factor memory location indicated by the addressing logic. Once the results are computed, they go through another exchange circuit and then to a demultiplexer that directs them to the correct destination. During FFT and IFFT calculation the results are written back into the internal memory for the next stage of the FFT. If the block is calculating an FFT for a code sequence, during the final stage the results are written into a separate code FFT memory whose function is to hold the finished results of the code FFT to be used for the complex conjugate multiplication. If the block is instead calculating an FFT for the signal sequence, the results are forwarded to the multiply block that performs the complex conjugate multiplication with corresponding samples from the code FFT memory. These results are then stored in the internal memory and this way the unit can start the IFFT operation using these values right away when the last FFT stage finishes. During the last stage of the IFFT calculation the results are forwarded to outside of the block and to the magnitude block.

The design is synthesized for 28 nm ASIC technology using 200 MHz clock frequency. This allows comparison with a pre-existing time-domain implementation that is used as a reference design for the results.

Since the system operates on a large amount of samples, one of the important characteristics of the design is how well it is able to pipeline the process. This ability is described in Catapult by the initiation interval, (*II*), value of a loop, which tells how many cycles it has to wait before it can start another iteration of the loop. For example to achieve the best performance, the *II* would need to be one, making it possible to start a new loop iteration on every clock cycle.

Most of the time spent for the correlation computation is spend inside the loop that goes through the butterfly operations. This means that the performance of this system is mainly dictated by the *II* of the butterfly loop. Performance-wise the ideal *II* for this would be one, so that new butterfly operation can be started on every clock cycle. On the other hand, *II* value of one would typically mean an increase in the area requirements of the design due to added pipeline registers. This also means that to achieve the *II* value of one, the system needs to be able to read inputs and write outputs at the same time. This requires the use of double port memory for the internal memory or alternatively adding another single port memory in the design and using sort of a ping-pong structure so the butterfly operation inputs are read from one memory and the results are written to the other one. After every stage the direction of data would change between the memories.

The magnitude block in the design, being a rather simple structure, can be pipelined with an interval of one and thus it isn't acting as bottleneck for the design. On the other hand

the current version of the FFT block implementation is not able to support this *II* value with the target clock frequency. The best result the design is able to achieve without lowering the clock speed is an II value of 3 for the butterfly loop. This makes the performance worse than the theoretical optimum, so further work is be needed to improve the design.

5.2.2 Mixed-radix-248

When it comes to calculating the FFT, radix-2 system is a rather simple design and especially since the in-place architecture is used, the speed of the computation is somewhat slow. In total, it's necessary to go through $N/2$ butterfly operations per stage and the number of stages is $log_2(N)$. One option to increase the speed would be to add another butterfly unit and calculate two butterflies in parallel, thus decreasing the number of required cycles in the loop by half. A better alternative is to use a higher radix butterfly to reduce the number of butterflies down to N/rad and the number of stages to $log_{rad}(N)$. But as mentioned earlier the transform length must be a power of the radix used, so by just using, for example, radix-4 butterfly, the ability to run FFT with lengths that are powers of two but not four is lost. A solution to this is a mixed-radix approach which allows the design to support lengths that are composites of the radixes-used. As an improved version of the design, a mixed-radix approach is used with a butterfly unit that supports radices 2, 4 and 8. Benefit of using these radices is that the butterfly unit can be constructed in a way that its data flow diagram resembles the radix-2 version. This can be seen in the following figure.

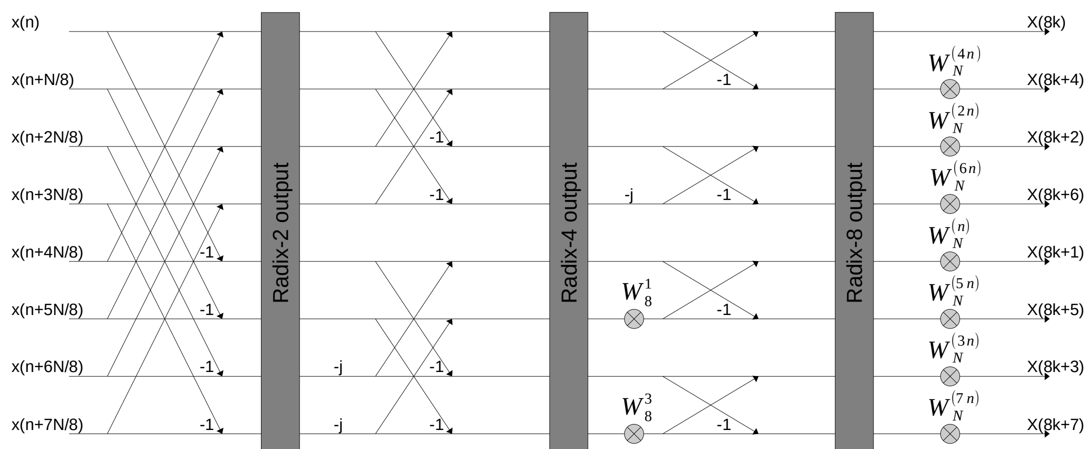

Figure 5.4. *Data flow diagram of a mixed-radix-248 butterfly*

In the figure 5.4 it's shown that the butterfly consists of multiple radix-2 butterfly structures in three different stages. The location of the output depends on the radix that is used. If radix-2 is used, only the first butterfly stage on the left is used and the two following stages are skipped. This way the unit computes four simultaneous radix-2 butterflies. Then again if radix-4 butterfly is to be used, the first two butterfly stages are calculated before skipping the last one. This totals in 2 parallel radix-4 butterflies. Finally when calculating radix-8 butterfly, the whole circuit is used and all three stages are computed before multiplying the values with twiddle factors and outputting the results. So no matter what radix is used, the butterfly always operates on eight samples.

One noteworthy aspect in the figure 5.4 is the number of twiddle factor multiplications. In total, the data flow diagram has nine complex multiplications so it's understandable that the hardware requirements are higher for this design than for the simple radix-2 one. Another aspect of the twiddle factors is the way to get them. Inside the third stage of the butterfly, there are two twiddle factor multiplications, but these twiddle factors are constant. For this reason it is possible, for example, to just store those values inside the butterfly. The remaining seven twiddle factors then again need to be brought from outside the butterfly. One way to deal with the twiddle factors would be to store them inside a ROM memory and fetch the needed values for every butterfly computation like is done in the radix-2 design. The problem with this option is the number of memory accesses. Since it's only possible to read one twiddle factor per cycle, it would take seven clock cycles to read all of the values required for one butterfly unit. This adds latency to the design but also sets a limit to high fast it's possible to pipeline the design. Another option is to calculate the twiddle factors during the computation. In the proposed radix-248 design, COordinate Rotation DIgital Computer (CORDIC) -based algorithm is used for calculating the twiddle factors. C++ function for this is included in the Algorithmic C libraries [15]. This twiddle factor generator can be pipelined better so it doesn't create a bottleneck for the rest of the design.

As the radix of the system is increased, the addressing scheme gets more complex. In [16] it is shown how a similar radix-2 design as in [7], is extended into a radix-4 algorithm. Now, the exchange circuit has four alternative operations and is controlled by signal that is two bits wide. Still the overall principle remains the same as the control signal is derived from the butterfly counter and the FFT stage. In similar manner the addresses for the four different banks are results of inverting bits of the butterfly counter value. This can be further extended to work with radix-8 when the memory is divided into eight blocks.

With radix-8 the exchange circuit has 8 possible operations. With control signal value of zero, the inputs retain their locations but the other control values results in shuffling of the inputs. The control signal values and the corresponding outputs of the shuffle circuits are visible in table 5.2.

Table 5.2. *Options for the shuffle circuit in the mixed-radix version*

Control	0	1	2	3	4	5	6	7
Input				**Output**				
0	0	1	2	3	4	5	6	7
1	1	0	3	2	5	4	7	6
2	2	3	0	1	6	7	4	5
3	3	2	1	0	7	6	5	4
4	4	5	6	7	0	1	2	3
5	5	4	7	6	1	0	3	2
6	6	7	4	5	2	3	0	1
7	7	6	5	4	3	2	1	0

Since there are eight options for the exchange circuit control, the required signal width is three bits. To get these bits it's necessary to look at a three bit window from the butterfly counter value. Just like in radix-2 version, when doing the FFT, the window starts from the most significant bit (MSB) of the butterfly counter and moves in the direction of the least significant bit (LSB). The window is shifted 3 bits for every stage. Once again the first and the last stage are exceptions and during the first stage the input controls are always zeroes and for the last stage the output exchange controls are zeroes.

Since the memory is divided into eight banks, it's necessary to calculate 8 addresses. Like in the radix-2 version, the address for the first memory bank is once again just the value of the butterfly counter, which runs from zero to $N/8 - 1$. The other addresses however are a bit more complicated to generate than in the simpler radix-2 architecture. All the seven other bank addresses are generated by xoring the butterfly counter value with their respective bit masks. This can be considered similar to the radix-2 in a way that in radix-2, the mask would start with zero values, and stage-by-stage one mask bit, starting from the most significant bit, is set to one so the xor operation with butterfly counter results in inverting the bits. With radix-8 the masks for different memory banks all differ from each other. Instead of setting every bit from zero to one, the bits are set inside a three bit window that is once again shifted over the butterfly counter one stage at a time. Now inside that three bit window, bits are set to one in a manner that the value of those three bits forms the index number of the memory bank. For example to get the address mask for the bank two, only the middle bit inside the window is set to one and the others remain zero. During the next stage, the window is shifted by three and once again the middle bit inside the window is set to one. Code example of this address generation

can be seen below.

```
1  //STAGE LOOP
2  STAGE_LOOP: for (int j= 0; j < MAX_NUMBER_OF_STAGES; j++){
3
4      ...
5
6      // Create masks for computing the indices for butterfly inputs
7      #pragma hls_unroll
8      CREATE_IDXMASK: for (int i = 0; i < 7; i++){
9          idxMask[i] = [0];
10         #pragma hls_pipeline_init_interval 1
11         for (int k = 0; k < MAX_NUMBER_OF_STAGES, k++){
12             if (k == stage){
13                 break;
14             }else{
15                 IdxMaskShift = bitsForBtrfIdx - 3*(k+1);
16                 ac_int<MAX_BITS_FOR_IDX-3, false> tmp
17                                 = ((i+1) << idxMaskShift);
18                 IdxMasks[i] = idxMasks[i] ^ tmp;
19             }
20         }
21     }
22
23     ...
24
25     //BUTTERFLY LOOP
26     BUTTERFLY_LOOP: for(int btrfly = 0;
27         btrfly < MAX_NUMBER_OF_BUTTERFLIES; btrfly++){
28         ...
29         // Get the indices for butterfly inputs
30         idx0 = btrfly;
31         idx1 = btrfly ^ idxMasks[0];
32         idx2 = btrfly ^ idxMasks[1];
33         idx3 = btrfly ^ idxMasks[2];
34         idx4 = btrfly ^ idxMasks[3];
35         idx5 = btrfly ^ idxMasks[4];
36         idx6 = btrfly ^ idxMasks[5];
37         idx7 = btrfly ^ idxMasks[6];
38         ...
39     }
```

```
40          ...
41  }
```

Program 5.2. *C++ code for radix-248 address generation*

The program 5.2 shows a part of the mixed-radix FFT code that computes the memory addresses for the butterfly inputs. The masks are generated during each of the stages. This process is described on rows 8-20. The "CREATE_IDXMASK" - loop is executed seven times since seven masks are needed to get all of the addresses. The pragma on the line 7 tells to the HLS tool to completely unroll this loop so each iteration is done simultaneously and hardware resources are generated for each of the loop iterations. The line 10 contains a statement that sets the *II* value of the loop to one, so the time between starting the loop iterations is one. Both of these thing could also be set inside Catapult HLS instead of writing them in the code. The initialization of most of the variables is done previously in the code, but on row 16 a tmp variable is created using the ac_int datatype. First inside the angle brackets is defined the bit width of the variable and the second parameter defines that it's an unsigned value.

In program 5.2 it can be seen how the masks are generated. The loop on row 11 is iterated as many times as is the current stage value. For each iteration, a variable ranging from one to seven is shifted and XORed with the mask value to get the new mask value. The variable represents the three bit window as its value is the same as the number of the memory bank for whom the mask is computed. Once this is done enough times, execution breaks out of the loop and the correct mask is obtained for the current stage. Then during every butterfly loop iteration these masks are XORed with the butterfly counter value to get the memory addresses. This can be seen on rows 30-37.

The implemented mixed-radix design maximizes the use of radix-8 butterfly so in the case that the transform length is a composite of different radices, the design uses radix-8 butterfly on all of the stages except one. During the last stage of FFT or the first stage of the IFFT, the design uses other radix than eight if necessary. This is the stage where all the twiddle factor values are one. Since shifting the window by three goes evenly only for transform lengths that are powers of eight, the window sort of moves over the edge of the butterfly counter during the last stage of FFT that is not for this length. In this case the lower window bits that go over the LSB are simply ignored.

Once again, the performance of the design is dictated by the pipeline capability of the butterfly unit loop. The higher radix significantly reduces the total number of stages and butterfly units, thus giving this version potential for a much faster transform. For mixed-radix architecture the target clock frequency is likewise 200 MHz. At the time of writing this thesis, the mixed-radix design is fully working and synthesizable, but room for improvements and optimizations remain. Thus the size and performance of the desing could better. The lowest working *II* value at the moment for the butterfly loop is four, which

could be lower. Another optimization area is the input and output. Since the radix-8 requires eight inputs to be read from the external memory, and eight values written to the output, this design suffers from the bottleneck caused by the external memories just like the the radix-2 version. To overcome this, the design simply copies the data samples to the internal memory before the start of the FFT loop. Similarly during the final stage of IFFT, the results are first stored in the internal memory, from where they would be written to the output in a separate loop. This solution doesn't affect the butterfly loop negatively but understandably adds extra clock cycles to the total cycle count.

6 RESULTS

Since the goal of the thesis is to explore the option of doing the correlation in frequency-domain, it is necessary to compare the new implementation to the existing time-domain one. In this chapter, the performance metrics as well as the results are presented and analyzed.

6.1 Verification methods

The determine the accuracy of the design, test signals and code sequences are generated with Matlab. The signals are represented as 5 bit integers and the values ranged from -15 to 15. The codes are sequences of ones and minus ones. The correlation results are generated by running the design on test data and then the results are compared to the reference results. Reference results for the accuracy comparison are computed with FFT based correlation using built-in functions in Matlab. For error metrics, root-mean-square-error, (RMSE), relative to the maximum correlation value is used, as well as relative maximum error. An example of the correlation with the design can be seen in the following figure.

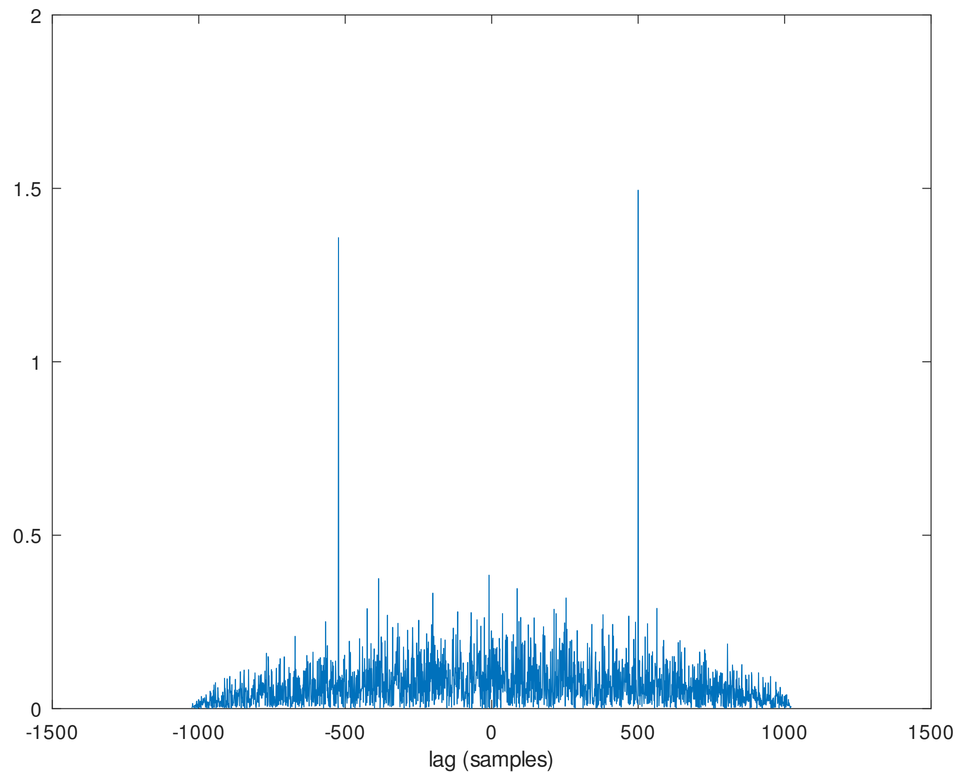

Figure 6.1. *Correlation result of the mixed-radix correlator for GPS signal*

In the figure 6.1 the output of the correlator design is shown. The signal used for the correlation was a GPS C/A with a code length of 1023 chips. It can be seen that one of the peaks is at the lag location of 500 samples, which is the delay value used for generating the test signal. The other peak is at -523 samples. This is due to the periodic nature of the PRN code.

As mentioned earlier, Xcelium simulation software is used to verify the design. It is also used to provide performance results for the system. Simulations are run with different signals and code lengths and from these runs the required cycle counts are extracted for comparison against the reference design.

The designs are synthesized for the 28nm ASIC technology using Cadence Genus Synthesis Solution software. These synthesis results provide the area scores for the designs. It's worth noting that the area scores are, to some degree, estimates, since due to problems with generating the technology libraries, the memories are not included in the synthesis. For this reason, the internal memories inside the FFT/ IFFT block are marked as external to the design in Catapult. This excludes them from the synthesis. A memory generator is used to provide the files necessary to create memory libraries in Catapult HLS and since the memory generator also gives area scores for memories, these values are added to the total area of the design.

6.2 Results comparison

In the following sections, results from the proposed designs are presented. This is followed by a comparison to the reference time-domain implementation.

6.2.1 Correlation error

Since the design uses fixed-point data types, the accuracy of the system is limited. As mentioned earlier, the limited number of bits reserved for the data causes overflow due to the addition operation in the butterfly unit. This problem is overcome by scaling the data. In this particular case right shift operation is used to do the scaling. This scaling combined with the twiddle factor multiplication as well as the complex conjugate multiplication causes instead problems in the other end of the bit sequence used to represent the data. Since only limited number of bits is reserved for the fractional part of the value, part of the value is lost to the truncation. Rounding operation is done before truncation to reduce this effect but the accuracy is still lower than it would be if the design wasn't restricted by limited bit width.

The accuracy of the correlation increases as the number of bits used for the data grows. On the other hand, larger bit width means larger area score for the design, especially due to the large memory usage. This means that the "ideal" bit width is a matter of compromise between area and accuracy. To determine the suitable number of bits, the design is compared to a reference frequency-domain correlation algorithm. The comparison is done using various GNSS signals over a range of different bit widths starting from 14 and ending at 32. From the total bit width, 5 bits are always reserved for the integer part and the rest is used for the fractional part. As mentioned, RMSE relative to the maximum value of the reference correlation as well as relative maximum error are used as metrics. In the following graph, a plot of the relative RMSE is shown.

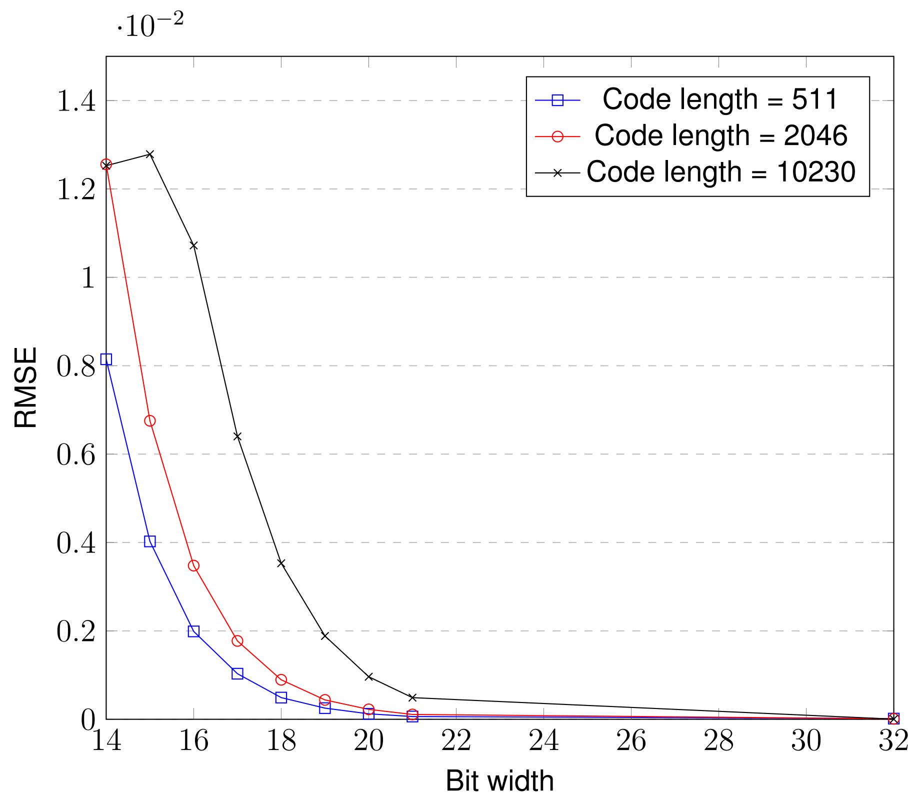

Figure 6.2. *Relative RMSE as a function of bit width*

In figure 6.2, error graphs for code lengths of 511, 2046 and 10230 are visible. The figures show that error decreases as the bit width increases and also that the error is larger for higher code lengths. This is due to the scaling done to avoid the overflow, and the zero padding caused by the linear correlation. The effect of zero padding increases as as the signal length grows since more zeroes need to be added. This affects especially code length of 10230 which requires zero padding to a length of 32768 so two thirds of the resulting sequence is actually just zeros. Combined with the scaling, the correlation results get rather small and more bits are required to represent those small fractions. The RMSE graph is actually a bit misleading since it shows quite small errors even for the lowest bit widths. This is because the reference frequency-domain correlation results calculated with Matlab are very close to zero, apart from the peak value. With too few bits, the small fractions can't be represented correctly and the output of the implemented correlator is close to zero. This happens to be close to the reference results, apart from the correlator peak. Maximum error shows this situation more accurately as can be seen in the following figure.

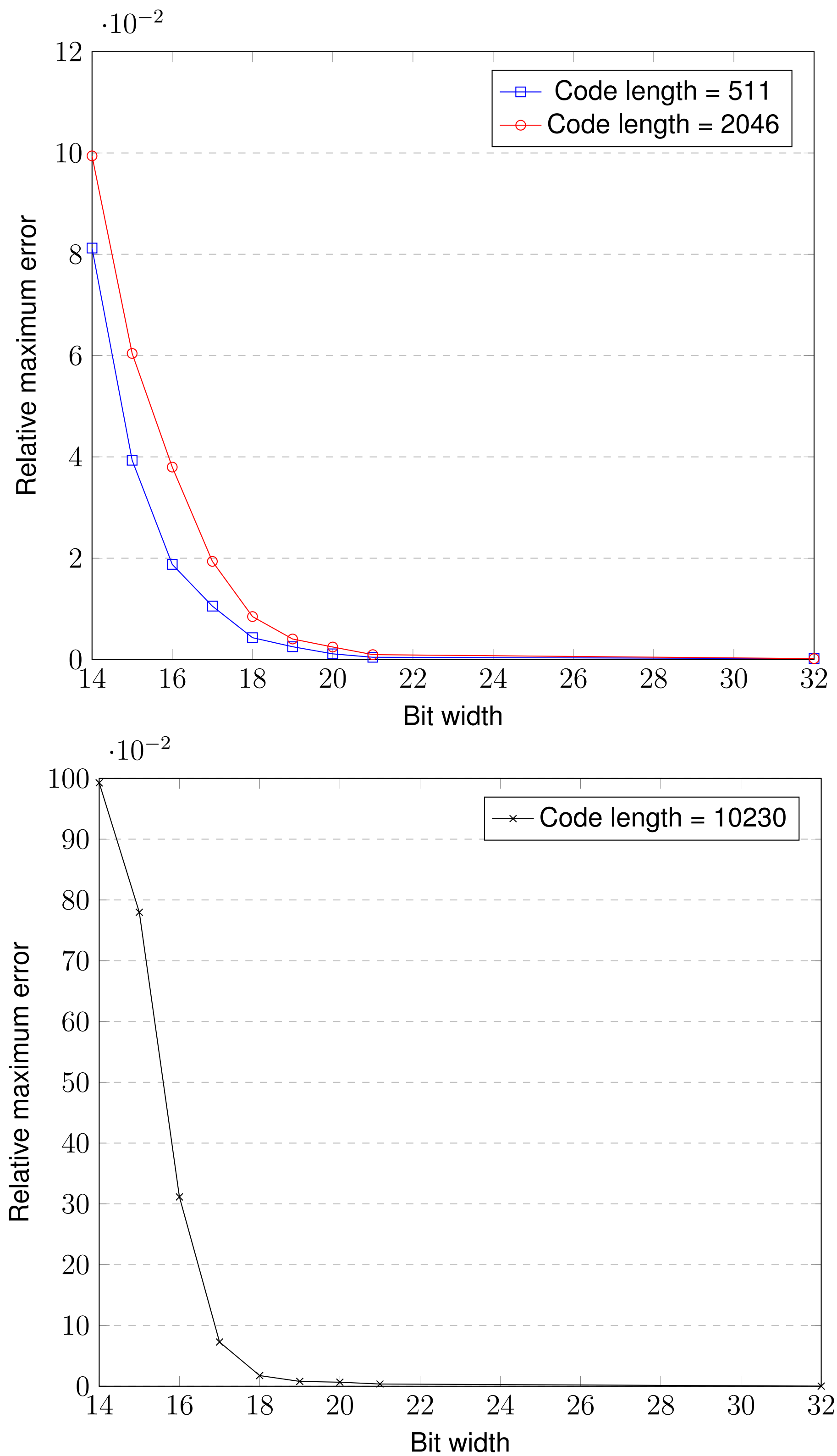

Figure 6.3. *Relative maximum error as a function of bit width*

Figure 6.3 shows that now for the code length of 10230 and with 14 bits, the maximum error relative to the peak value is very close to one. This effectively means that there is no peak and the results are incorrect. On the other hand the error decreases rapidly with

increasing bit width. On the right side, the bit width increase doesn't have as much of an impact anymore and further increasing the bit width wouldn't be worth the increased area. Once again it can be seen that the error is larger for longer sequences. If the goal is to achieve an error smaller than 1%, 18 bits would probably be suitable number.

6.2.2 Performance

Since the FFT algorithm makes it possible to reduce the number of arithmetic operations, there's a possibility to achieve performance gain over the time-domain implementation and reduce the time it's required to compute the correlation. To have an idea about performance of the implemented design, it's throughput is compared to an existing time-domain correlator version. Comparison is done for different GNSS code lengths. The cycle count results for the implementation are extracted from simulations and the reference number of clock cycles is approximated by the following equation 6.1,

$$\text{reference number of cycles} \approx c + \frac{c^2}{512}, \tag{6.1}$$

where c is the length of the code sequence. Due to the power of two in the numerator, the cycle count grows rapidly as the code length increases.

In the implemented design, most of the time spent on the correlation is spent for the FFT and IFFT operations. The complex conjugate multiplication and magnitude calculation add some latency to the design but most of the cycles are used inside the loop structures of the FFT/IFFT unit. For this reason, the required number of cycles for the design can be estimated with the number of butterfly calculations required to do the transform. Equation to get this number is shown below.

$$\text{butterfly calculations per FFT} = \text{number of stages} \cdot \text{butterfly calculations per stage} \tag{6.2a}$$

In 6.2a, the number of stages and the number of butterflies per stage depends on the radix in the following way.

$$\begin{aligned} \text{number of stages} &= \text{ceil}(\log_{\text{rad}}(N)) \\ \text{butterfly calculations per stage} &= \frac{N}{\text{rad}} \end{aligned} \tag{6.2b}$$

In 6.2b, N is the length of the FFT and the rad is the radix for the algorithm used. For mixed-radix algorithm, since the design uses radix-8 for all the stages except one, it can be chosen that $\text{rad} = 8$. The ceiling function is necessary in the case when N is not a

power of rad. With 6.2b, 6.2a can now be written as follows.

$$\text{butterfly calculations per FFT} = \text{ceil}(\log_{\text{rad}}(N)) \cdot \frac{N}{\text{rad}} \tag{6.2c}$$

Depending on the case, it might be necessary to calculate FFT for both the signal and the code sequences. It's also necessary to do IFFT operation, so altogether the required number of butterflies can be calculated from 6.2c by multiplying it by two or three, depending on the case. In the best-case scenario, the code FFT has already been done and the total number of butterfly calculations is

$$\begin{aligned} \text{total butterfly calculations} &= 2 \cdot \text{butterfly calculations per FFT} \\ &= 2 \cdot \text{ceil}(\log_{\text{rad}}(N)) \cdot \frac{N}{\text{rad}}. \end{aligned} \tag{6.2d}$$

The equation 6.2d can be further used to approximate the cycle count of the design. This is achieved by multiplying the total number of butterflies with the initiation interval. This results in

$$\text{number of cycles} \approx 2 \cdot \text{ceil}(\log_{\text{rad}}(N)) \cdot \frac{N}{\text{rad}} \cdot II, \tag{6.3}$$

where II is the initiation interval of the butterfly loop. As stated earlier, the ideal value is one so a butterfly is computed every clock cycle.

The following table contains the approximated reference number of cycles for different code lengths as well as the required cycles for the implementation gained from simulations. Number of butterflies, calculated with 6.2d, is also included to give some idea about the theoretical lower limit of the cycle count.

Table 6.1. *Required cycles for different code lengths*

Code length	511	1023	2046	4092	10230
FFT length	1024	2048	4096	8192	32768
Reference time-domain design	1021	3067	10222	36796	214630
Rad-2 butterflies	10240	22528	49152	106496	491520
Rad-2 cycles (II = 3)	30732	67596	147468	319500	1474572
Rad-248 butterflies	1024	2048	4096	10240	40960
Rad-248 cycles (II = 4)	6287	12431	24719	57527	229559
Rad-248 without copy (II = 4)	4239	8335	16527	41143	164023

In the two uppermost rows of table 6.1, there are code lengths and their respective transform lengths for linear correlation. Below that, there are approximated cycle counts for the reference time-domain design and then the cycle counts for the implemented designs. First for the radix-2 version and then for the mixed-radix version. The rows labeled with butterflies contain the number of butterfly calculations needed for the correlations and the rows labeled with cycles contain the cycle counts extracted from simulations. These numbers are for the best-case scenario when the code FFT has already been calculated.

Since the mixed-radix version does some extra work when it copies the input data into the internal memory before the FFT and writes the IFFT results into the memory before writing them to the output, the last row shows the cycle count with these extra input and output cycles subtracted. These values are calculated from the previous row, but with $2 \cdot N$ subtracted to remove the delay spent on that copying. Since the copy delay is due to the limitations of the external memory accesses and though it can be reduced by some design changes, it can only be completely ignored if the external memories are structured in a way that allows simultaneous reads and writes. This value is mainly to show the relation between the butterfly count and the actual cycles count.

By comparing the number of butterflies to the number of cycles for radix-2, and to the cycle count without the data copy for mixed-radix version, it can be seen that the estimation presented in equation 6.3 holds fairly well. The cycle count is roughly the same as the number of butterflies multiplied by the initiation interval, which is three for radix-2, and four for the mixed-radix version. The deviation from the estimate, especially with higher FFT lengths and mixed-radix cycle count, can be contributed to the overhead in the transform loop as well as to the added delay of the rest of the system.

As for the actual results, it can be seen that the radix-2 cycle count is significantly higher than for the reference model. Even if the ideal II value was achieved, the performance still wouldn't be on par with the reference. Adding another butterfly unit in the design would halve the cycle count but it still wouldn't be fast enough. So it is easy to see that a higher radix version of the FFT is needed.

When it comes to the mixed-radix version, it achieves better results than the radix-2. Unfortunately the time spent to compute the correlation is still longer than it is for the reference. On the other hand, the butterfly count gives some promising results and since the $II = 4$, the design has some room for improvement. The current implementation is already close to the throughput of the reference model for the highest transform length, so a reduction of II by one would be enough to make the design faster with that code length. Further reductions would make the design competitive even with lower code lengths. Getting the cycle count to match the butterfly count exactly is unfortunately beyond reach due to the limitations set by external memory accesses, so for the shortest code lengths, the time domain approach is going to be faster.

6.2.3 Design area

Another important metric for comparison is the area of the design. The area for the reference model is given as a single value and further analysis on how that area is allocated is left outside of this comparison. For the FFT correlator implementations, the total areas are given, as well as some breakdown on how the area is divided between logic and memories. The areas are extracted from synthesis results, with the exception of memories. As the memory synthesis for the ASIC technology turned out to be a little problematic, the memory areas are taken from a memory generator and added to the areas of rest of the designs. The area comparison is shown in the following table.

Table 6.2. *Area requirements for different designs*

Design	Total area	FFT logic	Magn. Block	SRAM	ROM	Twiddle Gen
Reference	78617	-	-	-	-	-
Rad-2	428058	13701	2939	394848	16559	-
Rad-248	638987	147748	4392	477469	-	38293

The table 6.2 shows the area breakdown for different implementations. The FFT implementations are generally larger than the time-domain version. Most of their area is in the FFT/IFFT block and especially in the memories. The memories can be divided into Static Random Access Memories (SRAM) and Read Only Memories (ROM). SRAM is used for the internal memories for intermediate values as well as to hold the code FFT results. ROM is used for storing the twiddle factors. If memories are ignored, the radix-2 version is actually relatively small, as can be seen from the FFT logic column. Size of the magnitude block is small compared to the FFT unit and this is easy to understand due to its limited functionality and simple structure.

The mixed-radix version is significantly larger than its radix-2 counterpart. Large portion of this area increase is due to the increased size of the logic. The logic area is roughly ten times the size of the corresponding radix-2. This is due to the more complex address generation logic and increased number of registers since now it's necessary to hold eight values and addresses instead of two. Another contributing factor to the area increase are the twiddle factors. As mentioned earlier, this version doesn't have a ROM memory to store the twiddle factors. Instead, it utilizes a CORDIC based algorithm to calculate the twiddle factors during the computation. This logic is multiplied seven times to get all the twiddle factors at one, which increases the area.

The total storage space for the two implementations is the same, but the partitioning of the memories is dependent on the radix. For radix-2 the memory is split into two

banks whereas for the mixed-radix version the memories are divided into eight parts. This explains the difference in memory areas between radix-2 and radix-248 versions.

It is also worth noting that these FFT correlator area results are for designs with a bit width of 16. Increasing the bit width leads to a larger area, especially due to the increase in memory size.

7 CONCLUSION AND FUTURE WORK

As the code lengths in GNSS systems grow, there is demand for faster alternatives for calculating the correlation between the signal and the code. Frequency-domain correlation offers an alternative solution to the more traditional time-domain approach. Although the frequency-domain version requires multiple DFTs and IDFTs to compute the correlation, FFT algorithm provides an overall reduction in number of arithmetic operations. The computational benefit increases as the code length grows, giving an option worth considering.

The characteristics of an FFT based correlator depend mainly on the implementation of the FFT engine. Two fundametal alternatives for this are pipelined FFT and in-place FFT. Generally the pipelined version offers much better throughput, but it's necessary to have multiple computational units, which increases the area of the design. In-place algorithms on the other hand typically rely on a single butterfly unit to operate on data that is located in one memory. Some adjustments to the memory structures and number of butterflies are still possible. Since the number of processing elements is smaller, this results in smaller overall area of the design but the performance is worse than with the pipelined design. In the end it comes down to a compromise between size and performance. Since the memory requirements alone make the design rather large, the in-place algorithm is often chosen, as is done in this thesis.

The results for the work proved quite promising. Large number of complex multiplications and necessary scaling of the results leads to an inherent error in the correlation results, but with right design choices this error can be made manageable. As for the performance, the implementations done in this thesis proved out to be insufficient compared to the highly optimized reference time-domain design. A radix-2 architecture is not fast enough and higher radix is needed. The radix-248 implementation provides better results but the performance was still lower than for the reference model. The biggest problem with the design was the suboptimal pipelining of the butterfly loop structure. Some hope is given by the fact that in this regard there is room for improvement. As shown by the butterfly count in table 6.1, the mixed-radix version has potential to outperform the reference at least with the higher code lengths. With the code length of 10230 the potential amount of saved cycles is quite significant.

As for the area of the design, it was clear from the beginning that, due to the nature of the algorithm, the size is going to be quite large. This is mainly due to the memories that are needed to hold the intermediate results during the FFT and IFFT operations, as well as to store the FFT results of the code sequence. Since it's necessary to store the samples somewhere, it's not really possible to reduce the storage requirements. Although, with some bit width optimizations it might be possible to achieve some area reduction.

The high level synthesis workflow proved to be helpful throughout the process. The biggest problems I experienced during the work were related to the FFT algorithm. While the HLS couldn't really help me in that regard, what it was able to do, was to let me focus on the important parts. Instead of spending my time on VHDL code and all its fine details, I was able to focus on the algorithm side. Naturally new tools include a little bit of a learning curve and it's going to take some time and work to be able to fully grasp all the intricacies of HLS and Catapult to get the optimal results. Especially since it wasn't always that clear, in what kind of hardware certain code structures would result. Overall, the tools and higher abstraction layer made it possible to easily test different kinds of architectures, with a small cost of losing some level of control over the final design. For example once the mixed-radix address generation logic was figured out, changing the radix-2 design to mixed-radix one was relatively easy.

When it comes to possible future work, there's quite a lot that can be done. The hardware implementation of the FFT alone has room for improvements to get the throughput closer to the theoretical optimum for this kind of architecture. Another area that could use some work is the FFT algorithm itself. As mentioned earlier, there are lot of options for the architecture, for example the radix of the butterfly as well as the address generation logic.

Significant reduction to the design area can be achieved by adding support for radix-5. The linear correlation has a requirement of having the transform length of $N + M - 1$, where N and M are the lengths of the signal and the code. Since in this case, those lengths are the same, the FFT length needs to be twice as long as the code. Another requirement is set by the FFT, as the transform length needs to be a composite of the radices used. This causes problems with the longest code as the linear correlation requires an FFT length of 20460, but due to usage of radix-2 it's necessary to zero pad the signal all the way to 32768. This means that one third of the zero padding is basically unnecessary. If radix-5 was also supported in the mixed-radix algorithm, it would be possible to use a length of just 20480. This would yield significant reductions in both memory size as well as in required cycles for the highest code length, thus improving the design. However implementing this radix-5 support isn't really a trivial task and the FFT/IFFT block would most likely require significant changes since the current address generation is designed for power-of-two radixes.

REFERENCES

[1] Kaplan, E. D. and Hegarty, C. J. *Understanding GPS/GNSS : principles and applications*. eng. Third edition. GNSS Technology and Applications Series. Boston ; Artech House, 2017. ISBN: 1-5231-1769-9.

[2] Groves, P. D. *Principles of GNSS, Inertial, and Multisensor Integrated Navigation Systems*. eng. Norwood: Artech House, 2013. ISBN: 1608070050.

[3] Tsui, J. B.-y. *Fundamentals of global positioning system receivers a software approach*. eng. Wiley series in microwave and optical engineering. New York: John Wiley & Sons, 2000. ISBN: 1-280-36745-8.

[4] Rao, K. *Fast Fourier Transform - Algorithms and Applications*. eng. 1st ed. 2010. Signals and Communication Technology. Dordrecht: Springer Netherlands, 2010. ISBN: 1-4020-6629-5.

[5] Cooley, J. W. and Tukey, J. W. An algorithm for the machine calculation of complex fourier series. eng. *Mathematics of computation* 19.90 (1965), 249–259. ISSN: 0025-5718.

[6] Polychronakis, N., Reisis, D. and Tsilis, E. A continuous-flow, Variable-Length FFT SDF architecture. eng. *2010 17th IEEE International Conference on Electronics, Circuits and Systems*. IEEE, 2010, 730–733. ISBN: 1424481554.

[7] Xiao, X., Oruklu, E. and Saniie, J. An Efficient FFT Engine With Reduced Addressing Logic. eng. *IEEE transactions on circuits and systems. II, Express briefs* 55.11 (2008), 1149–1153. ISSN: 1549-7747.

[8] Fingeroff, M. *High-level synthesis: blue book*. eng. Xlibris Corporation, 2010. ISBN: 978-1-4500-9724-6.

[9] *Catapult High-Level Synthesis and Verification*. en. URL: https://eda.sw.siemens.com/en-US/ic/catapult-high-level-synthesis/ (visited on 04/11/2021).

[10] *Vivado Design Suite*. en. URL: https://www.xilinx.com/products/design-tools/vivado.html (visited on 04/11/2021).

[11] *High-Level Synthesis Compiler - Intel® HLS Compiler*. en. URL: https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html (visited on 04/11/2021).

[12] *Stratus High-Level Synthesis*. en. URL: https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/synthesis/stratus-high-level-synthesis.html (visited on 04/11/2021).

[13] *HDL Coder*. en. URL: https://www.mathworks.com/products/hdl-coder.html (visited on 04/11/2021).

[14] *Genus Synthesis Solution*. ko. URL: https://www.cadence.com/ko_KR/home/tools/digital-design-and-signoff/synthesis/genus-synthesis-solution.html (visited on 04/11/2021).

[15] *HLS Libs Website*. en. URL: https://hlslibs.org/ (visited on 11/15/2020).

[16] Long, S.-S., Hong, M.-Y. and Shiue, M. T. A Low-Complexity Generalized Memory Addressing Scheme for Continuous-Flow Fast Fourier Transform. eng. *2018 3rd International Conference on Computer and Communication Systems (ICCCS)*. IEEE, 2018, 492–496. ISBN: 9781538663509.