Implementation of LTE baseband algorithms for a highly parallel DSP platform

Examensarbete utfört i Datorteknik vid Tekniska högskolan vid Linköpings universitet
av
Markus Keller

LiTH-ISY-EX--16/4941--SE
Linköping 2016
Implementation of LTE baseband algorithms for a highly parallel DSP platform

Examensarbete utfört i Datorteknik vid Tekniska högskolan i Linköping av

Markus Keller

LiTH-ISY-EX--16/4941--SE

Handledare:  Dake Liu
              ISY, Linköpings universitet

            Di Wu
              ISY, Linköpings universitet

Examinator:  Andreas Ehliar
              ISY, Linköpings universitet

Linköping, 25 April, 2016
The division of computer engineering at Linköping’s university is currently developing an innovative parallel DSP processor architecture called ePUMA. One possible future purpose of the ePUMA that has been thought of is to implement it in base stations for mobile communication. In order to investigate the performance and potential of the ePUMA as a processing unit in base stations, a model of the LTE physical layer uplink receiving chain has been simulated in Matlab and then partially mapped onto the ePUMA processor.

The project work included research and understanding of the LTE standard and simulating the uplink processing chain in Matlab for a transmission bandwidth of 5 MHz. Major tasks of the DSP implementation included the development of a 300-point FFT algorithm and a channel equalization algorithm for the SIMD units of the ePUMA platform. This thesis provides the reader with an introduction to the LTE standard as well as an introduction to the ePUMA processor. Furthermore, it can serve as a guidance to develop mixed point radix FFTs in general or the 300 point FFT in specific and can help with a basic understanding of channel equalization. The work of the thesis included the whole developing chain from understanding the algorithms, simplifying and mapping them onto a DSP platform, and testing and verification of the results.
Abstract

The division of computer engineering at Linköping’s university is currently developing an innovative parallel DSP processor architecture called ePUMA. One possible future purpose of the ePUMA that has been thought of is to implement it in base stations for mobile communication. In order to investigate the performance and potential of the ePUMA as a processing unit in base stations, a model of the LTE physical layer uplink receiving chain has been simulated in Matlab and then partially mapped onto the ePUMA processor.

The project work included research and understanding of the LTE standard and simulating the uplink processing chain in Matlab for a transmission bandwidth of 5 MHz. Major tasks of the DSP implementation included the development of a 300-point FFT algorithm and a channel equalization algorithm for the SIMD units of the ePUMA platform. This thesis provides the reader with an introduction to the LTE standard as well as an introduction to the ePUMA processor. Furthermore, it can serve as a guidance to develop mixed point radix FFTs in general or the 300 point FFT in specific and can help with a basic understanding of channel equalization. The work of the thesis included the whole developing chain from understanding the algorithms, simplifying and mapping them onto a DSP platform, and testing and verification of the results.
Acknowledgments

First of all, I would like to thank both professor Dake Liu for the opportunity to work with this interesting thesis topic as well as my examiner Andreas Ehliar for his valuable input regarding my thesis report. Also I would like to thank Andreas Karlsson for his helpful support with any questions I had regarding programming the sleipnir processor and Di Wu for his support with LTE related questions.

Another thanks goes to my opponent Mohamed Lababidi for his comments regarding my thesis, which helped me to improve the quality of the report.

Last but not least I would like to thank my parents for giving me the opportunity to study abroad and for their love, support and patience.
## Contents

1 Introduction ......................................................... 3
   1.1 Background .................................................. 3
   1.2 Scope .......................................................... 3
   1.3 Outline ....................................................... 4

2 LTE, Long Term Evolution ........................................... 7
   2.1 Background .................................................... 7
   2.2 Design targets ................................................ 8
      2.2.1 Capabilities .............................................. 8
      2.2.2 System performance ..................................... 8
      2.2.3 Deployment-related aspects ............................. 9
      2.2.4 Strategies to meet the design targets .................. 9
   2.3 OFDM ............................................................ 10
      2.3.1 Multi-carrier transmission ............................. 10
      2.3.2 Physical Resource ....................................... 11
      2.3.3 Orthogonality ............................................ 12
      2.3.4 Cyclic prefix ............................................. 13
      2.3.5 Modulation using FFT processing ....................... 13
      2.3.6 Advantages and drawbacks .............................. 16
   2.4 SC-FDMA ........................................................ 17
      2.4.1 DFT-spread OFDM processing chain ..................... 17
   2.5 Frame structure ............................................... 18
   2.6 Reference signals ............................................ 20
      2.6.1 Downlink reference signals ............................ 21
      2.6.2 Uplink reference signals ............................... 22

3 Project implementation ............................................. 25
   3.1 Project overview ............................................. 25
      3.1.1 General assumptions ..................................... 25
      3.1.2 Transmission bandwidths ............................... 26
      3.1.3 The uplink receiving processing chain ................ 26
   3.2 Channel estimation and signal equalization .................. 27
      3.2.1 Channel estimation ...................................... 28
      3.2.2 Channel equalization with Zero forcing ............... 29
3.3 Implementation in Matlab ........................................ 29

4 The ePuma DSP platform ........................................... 33
  4.1 Background ...................................................... 33
  4.2 The ePUMA architecture ......................................... 34
  4.3 Memory subsystem and data communication ...................... 35
    4.3.1 Memory hierarchy ........................................ 36
    4.3.2 Data communication ...................................... 37
  4.4 The ePUMA Master ............................................. 38
  4.5 The SIMD processors .......................................... 39
    4.5.1 Supported datatypes ...................................... 39
    4.5.2 Sleipnir datapath ......................................... 40
    4.5.3 Memory components and registers ......................... 41
    4.5.4 Data access of LVMs and vector register file ............. 41
    4.5.5 Programming the sleipnir ................................ 43

5 FFT implementation ................................................. 51
  5.1 DFT and IDFT relation ......................................... 51
  5.2 Fast Fourier Transformation ................................... 53
    5.2.1 Cooley-Tukey FFT algorithm .............................. 53
    5.2.2 Radix-4 and Radix-2 FFTs ............................... 58
  5.3 300 point FFT Implementation for Sleipnir .................... 58
    5.3.1 Cycle cost estimations ................................... 59
    5.3.2 Assembly implementation ................................ 61
    5.3.3 Cycle costs ................................................ 63

6 Implementing the channel equalization .......................... 65
  6.1 Implementing the computation of the channel estimate .......... 65
  6.2 Implementing the zero-forcing equalizer ....................... 66
    6.2.1 Kernel for 1/x .......................................... 68
    6.2.2 Assembly code and cycle cost ............................ 68
  6.3 Cycle cost ...................................................... 71

7 Results and discussion ............................................. 73
  7.1 Cycle cost of FFT algorithm .................................. 73
  7.2 Evaluation of the results ..................................... 74
  7.3 Challenges of the implementation ............................. 76

8 Conclusions .......................................................... 79
  8.1 Future work .................................................... 80

Bibliography .......................................................... 83

A Kernel for 300 point FFT ........................................... 85
# List of abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>3G</td>
<td>Third generation</td>
</tr>
<tr>
<td>3GPP</td>
<td>Third generation partnership project</td>
</tr>
<tr>
<td>ALU</td>
<td>Arithmetic logic unit</td>
</tr>
<tr>
<td>BS</td>
<td>Base station</td>
</tr>
<tr>
<td>BW</td>
<td>Bandwidth</td>
</tr>
<tr>
<td>CM</td>
<td>Constant memory</td>
</tr>
<tr>
<td>CP</td>
<td>Cyclic prefix</td>
</tr>
<tr>
<td>CPU</td>
<td>Central processing unit</td>
</tr>
<tr>
<td>D/A</td>
<td>Digital to analog</td>
</tr>
<tr>
<td>DAC</td>
<td>Digital to analog converter</td>
</tr>
<tr>
<td>DL</td>
<td>Downlink</td>
</tr>
<tr>
<td>DFT</td>
<td>Discrete fourier transform</td>
</tr>
<tr>
<td>DMA</td>
<td>Direct memory access</td>
</tr>
<tr>
<td>DFTS-OFDM</td>
<td>DFT spread OFDM</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital signal processor/processing</td>
</tr>
<tr>
<td>ePUMA</td>
<td>Embedded parallel DSP processor with Unique Memory Access</td>
</tr>
<tr>
<td>FDD</td>
<td>Frequency division duplex</td>
</tr>
<tr>
<td>FFT</td>
<td>Fast fourier transform</td>
</tr>
<tr>
<td>HSPA</td>
<td>High speed packet access</td>
</tr>
<tr>
<td>IDFT</td>
<td>Inverse discrete fourier transform</td>
</tr>
<tr>
<td>IFFT</td>
<td>Inverse fast fourier transform</td>
</tr>
<tr>
<td>ISY</td>
<td>Institutionen för systemteknik</td>
</tr>
<tr>
<td>LTE</td>
<td>Long term evolution</td>
</tr>
<tr>
<td>LVM</td>
<td>Local vector memory</td>
</tr>
<tr>
<td>MBMS</td>
<td>Multimedia broadcast multicast service</td>
</tr>
<tr>
<td>MIMO</td>
<td>Multiple input multiple output</td>
</tr>
<tr>
<td>MMSE</td>
<td>Minimum Mean Square Error</td>
</tr>
<tr>
<td>PM</td>
<td>Program memory</td>
</tr>
<tr>
<td>PPU</td>
<td>Packet processing unit</td>
</tr>
<tr>
<td>OFDM</td>
<td>Orthogonal frequency-division multiplexing</td>
</tr>
<tr>
<td>QAM</td>
<td>Quadrature amplitude modulation</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------</td>
</tr>
<tr>
<td>QPSK</td>
<td>Quadrature phase-shift keying</td>
</tr>
<tr>
<td>RB</td>
<td>Resource block</td>
</tr>
<tr>
<td>RF</td>
<td>Radio frequency</td>
</tr>
<tr>
<td>RISC</td>
<td>Reduced instruction set computing</td>
</tr>
<tr>
<td>SC-FDMA</td>
<td>Single carrier frequency division multiple access</td>
</tr>
<tr>
<td>SIMD</td>
<td>Single instruction multiple datapath</td>
</tr>
<tr>
<td>TDD</td>
<td>Time division duplex</td>
</tr>
<tr>
<td>UL</td>
<td>Uplink</td>
</tr>
<tr>
<td>VRF</td>
<td>Vector register file</td>
</tr>
<tr>
<td>WCDMA</td>
<td>Wideband code division multiple access</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Background

Recent trends in the development of computer technology show an increasing importance of parallel computing methods to fulfill the ever rising demands on computational performance. With this development trend in line, the division of computer engineering at Linköping’s university is currently developing a highly parallel DSP platform called ePUMA. The platform consists of a master-processor and eight SIMD co-processors called sleipnir, which are capable of 128-bit vector operations. By combining the multi-core processor design approach with the SIMD approach, an embedded DSP platform capable of high performance parallel computing shall be created.

The ePUMA is considered to be used in various DSP fields like, for example, baseband signal and radar signal processing, video games and video coding and decoding [12]. This thesis will investigate the ePUMA’s potential of serving as a processing unit in base stations supporting LTE, the latest developed standard for mobile communication. For this purpose, a model of the LTE uplink communication has been developed and parts of the processing chain have been mapped onto the ePUMA.

1.2 Scope

In order to achieve the goal of implementing LTE-baseband algorithms for the ePUMA, the software Matlab has been used as a tool for implementing a simplified model of the complete LTE physical layer processing chain at the uplink receiver side. The communication was modeled for a transmission bandwidth of 5 MHz, for which the LTE baseband processing includes the computation of a 300 point IFFT computation. For the ePUMA platform, the main focus was to implement certain algorithms which were included in this processing chain for the SIMD sleipnir co-processors. The implementations that have been made for the co-processors are a 300-point FFT computation and a channel equalization algo-
rithm. This thesis provides the theoretical background for understanding the LTE baseband processing that has been modeled in Matlab and for understanding the algorithm implementations that have been made for the ePUMA. Since the implementation of the baseband uplink receiving chain is a challenging task, a highly simplified model of the uplink LTE baseband receiver has been used as a basis for the implementations made in this thesis.

1.3 Outline

The thesis is organized into the following chapters:

- **Chapter 2 - LTE Introduction** gives an introduction to the LTE standard. The chapter begins with a discussion of the driving forces most relevant for the market of mobile communications and explains to which LTE design targets these driving forces have led. Based on that, the reader can get a deeper understanding of the motivations for using certain technologies in the LTE standard. The chapter mainly focuses on aspects that help understanding the LTE physical layer processing and the baseband algorithms that were implemented in this thesis. In particular it will be clarified why the LTE baseband processing includes FFT computations and how the LTE standard defines reference symbols that can be used for channel equalization.

- **Chapter 3 - Project Implementation** presents an overview of the project and the scope of the implementations that have been made in Matlab and for the ePUMA. With the basic knowledge about the LTE standard discussed in the previous chapter, we will have a basis to understand the model of the physical layer LTE uplink chain that has been implemented. This chapter discusses which assumptions and simplifications have been made to derive the Matlab model of the LTE uplink baseband processing at the receiver side. Furthermore, it will be clarified which parts of the uplink processing chain have been mapped onto the ePUMA.

- **Chapter 4 - The ePUMA platform** gives an introduction to the ePUMA DSP platform. This chapter discusses why the concept of the ePUMA platform of utilizing different forms of computing parallelism is coherent with recent trends in processor technologies. Furthermore, it provides the reader with an overview of the architecture of the platform. The main focus of the chapter however, is to explain how the eight SIMD processors of the ePUMA are programmable in order to enable the reader to understand the implementations that have been made in this thesis.

- **Chapter 5 - Implementing 300 point FFT** provides a detailed description of how the 300 point FFT has been implemented for the SIMD
processors of the ePUMA. The chapter begins with presenting the theoretical background that is the base for deriving the FFT algorithm which has been used. Furthermore, it will be shown which minimum cycle cost is theoretically achievable with the approach used for the algorithm implementation in this work.

- **Chapter 6 - Implementing the channel equalization** This chapter describes how the channel equalization has been implemented for the SIMD processors of the ePUMA, and which cycle cost the execution of the algorithm requires.

- **Chapter 7 - Results and discussion** Presents the results of the thesis. The chapter discusses the cycle cost that has been achieved and the challenges that have been faced during the implementation. Based on that, some potential improvements for the SIMD co-processors are suggested.

- **Chapter 8 - Conclusion** The last chapter concludes the thesis with a summary of the work, a final discussion about ePUMA’s potential as a processing unit in base stations, and suggestions for future work.
Chapter 2

LTE, Long Term Evolution

2.1 Background

The demands on mobile communication systems are continuously growing, due to the rapidly increasing numbers of mobile users worldwide and a hard competition between various existing and new network operators and vendors [4]. This competition leads to a constant development of new standards and technologies in order to provide new services for the mobile users as well as existing services in better ways and at a lower cost [4]. More and more people are interested in broadband data access everywhere in order to use services like email-synchronization, internet access, file download, video streaming, teleconferencing and other specific applications for their mobile devices [10],[14].

Due to the challenge of increasing numbers of mobile users on the one hand and their requests for more advanced services on the other hand, the requirements on advanced mobile systems of today are very widely spread and include high demands on service data rates, user throughput, mobility, cell coverage, spectrum efficiency, spectrum flexibility, system complexity and many more [3]. Therefore, developing mobile systems has become a very complex and challenging task, which is being carried out by global standard-developing organizations such as the Third Generation Partnership Project (3GPP) including thousands of people [4]. With over 3 billion users, the mobile system technologies specified by 3GPP are the most widely distributed in the world and the partners of the project include organizations from China, Europe, Japan, Korea and the USA [4]. Long Term Evolution (LTE) is the latest standard for mobile communication systems developed by 3GPP and was first launched for commercial use in December 2009 in Stockholm and Oslo. The number of users have been continuously growing since and in 2015 LTE has been launched in 93 countries while the total number of LTE subscription was reported to have reached the 500 million mark.
2.2 Design targets

In the following some important design targets of LTE will be explained. We should note here, that all of those design targets have even been surpassed by the current performance of LTE. The LTE requirements can be divided into seven different categories [4]:

- **Capabilities**: Uplink and downlink data-rates
- **System performance**: User throughput, spectrum efficiency, mobility and coverage
- **Deployment related aspects**: Spectrum flexibility, spectrum deployment, and coexistence with other 3GPP standards
- **Architecture and migration**: LTE Radio Access Network architecture design targets
- **Radio resource management**: Specification for support of higher-layer transmission and for load sharing and authority management between different radio access technologies
- **Complexity**: Overall system-complexity as well as complexity of the mobile terminal
- **General aspects**: Cost and service related aspects

We will discuss the first three categories in more detail, as it will help to understand the motivation for implementing certain technologies in LTE, which are relevant for the algorithms implemented in this thesis.

2.2.1 Capabilities

The LTE standard offers peak data rates which are around ten times higher than previous third generation (3G) standards. For the downlink (DL) transmission, that means from the base stations (BS) to the mobile user, data rates up to 100 Mbit/s can be achieved in the maximum supported transmission bandwidth of 20 MHz[3]. The uplink (UL) communication (from mobile user to BS) offers data rates up to 50 Mbit/s for this bandwidth [3]. Decreasing the transmission bandwidth will proportionally decrease the peak data rate and thus we can express the LTE data rate requirements as 5 bit/s/Hz for the DL and 2.5 bit/s/Hz for the UL [4].

2.2.2 System performance

The system performance specifications define the targets for user throughput, spectrum efficiency, mobility, coverage and Multimedia Broadcast or Multicast Services (MBMS). In the LTE specifications these design targets are defined in comparison to an earlier 3GPP standard namely Release 6 HSPA [4].
2.2 Design targets

User throughput describes how many mobile users can be served simultaneously by the LTE system in a certain cell, or in other words, in the coverage area of a certain base station. Compared to the earlier HSPA standard, the average user throughput is aimed to be 3 to 4 times higher for DL and 2 to 3 times higher for UL.[4]

Spectrum efficiency describes the system throughput within one cell per bandwidth and is thus measured in bit/s/MHz/cell. Similar to the average user throughput, the LTE spectrum efficiency shall be improved around 3 to 4 times for DL and 2 to 3 times for UL. [4]

Mobility requirements address the mobile terminals’ speed. According to the design targets, maximum performance can be achieved for low mobile terminal speeds between 0-15 km/h and for speeds up to 120 km/h LTE is still able to provide a relatively high performance. The maximum speed of a mobile terminal that can be handled by LTE depends on the frequency bands and lies within 350 to 500 km/h [4].

Coverage requirements deal with the cell radius, which defines the maximum possible distance between a base station and the mobile terminal connected to the base station. The requirements for the user throughput, spectrum efficiency and mobility mentioned above shall be achieved for a cell radius up to 5 km, while a slight decrease in performance is tolerated for a cell range up to 30 km [4].

MBMS requirements deal with the so called point to multipoint services that allow sending the same data to many mobile users simultaneously. Possible MBMS services could include for example mobile TV or radio broadcasting and shall have a higher performance than in previous standards. The design target for the spectral efficiency of MBMS services is defined as 1 bit/s/Hz [4].

2.2.3 Deployment-related aspects

The deployment-related aspects deal mainly with the frequencies used by the LTE system. Since it should be possible to install LTE step by step in frequency bands which are already allocated by other 2G or 3G standards, the requirements on the spectrum flexibility of LTE are very high. The specifications define that LTE should support both TDD and FDD which makes it possible that LTE can be used both in paired and unpaired spectrum allocations [10]. Furthermore LTE should be usable in different frequency bands and support scalable transmission bandwidths. The standard includes 6 different transmission bandwidths which range from 1.4 MHz up to 20 MHz[10].

2.2.4 Strategies to meet the design targets

We can summarize here that some of the main design targets of LTE are a high system performance in terms of data rate, spectrum efficiency and user throughput on the one hand, as well as high spectrum flexibility in order to make LTE
compatible with other already existing mobile systems on the other hand. The technical innovations included in LTE are enormous and it is beyond the scope of this thesis to discuss all of them in detail. Instead we will focus on some aspects of the LTE standard, which are important to meet the ambitious design targets, and which will help us to understand the uplink processing algorithms developed for the ePUMA processor in this thesis work.

Two of the key technologies for LTE to achieve the design targets mentioned above are multi antenna techniques and Orthogonal Frequency Division Multiplexing (OFDM)[10]. Multi antenna techniques have already been widely used in former mobile standards. The usage of more than one antenna at transmitter and/or receiver side can significantly increase the system performance in respect to user throughput, coverage, and data rates. Despite of the importance of multi antenna techniques for the LTE standard, they are somewhat out of the scope of this thesis. In the following section we will describe OFDM, which is another very important technique of LTE and more relevant for the algorithms implemented in this thesis.

2.3 OFDM

OFDM has been adopted for LTE downlink communication after it had previously successfully been used for several other technologies including for example digital audio and video broadcasting for local area networks [17]. It offers several benefits for achieving the LTE design targets including a high spectral efficiency and robustness against frequency selective fading caused by multipath propagation [3].

2.3.1 Multi-carrier transmission

OFDM is a special form of a multi-carrier transmission scheme. In a multi-carrier transmission the data that shall be transmitted is divided into several smaller data streams, which are then being transmitted in parallel and on different frequency bands as illustrated in Figure 2.1 [3]. These signals with a smaller bandwidth are often called subcarriers.

![Figure 2.1. Dividing data stream on multiple carriers](image)

The input $X_k$ represents a data stream of complex modulated symbols from
any complex alphabet like for example QPSK. Depending on the channel quality the transmission is either performed with modulation schemes corresponding to higher data rates (such as 64QAM), or with modulation schemes offering lower data rates (such as QPSK)[2]. Since the data streams are transmitted over different carrier frequencies they can be restored at the receiver side by the use of correlators [3]. Figure 2.2 [3] illustrates the corresponding spectrum of the multicarrier transmission with four subcarriers, considered in our example.

One of the main differences between OFDM and other multi-carrier transmission schemes used in previous standards like for example in Wideband Code Division Multiple Access (WCDMA) is that OFDM uses a significantly higher number of subcarriers which allocate relatively small frequency bands. A transmission bandwidth of 20 MHz for example would be equivalent to four subcarriers with approximately 5MHz bandwidth each in WCDMA, but in LTE the same overall transmission bandwidth consists of 1200 subcarriers [4]. Another difference be-

![Figure 2.2. Spectrum multi-carrier transmission](image)

 tween OFDM and ordinary multi-carrier transmission is that in the later one the spectra of the different subcarriers should not overlap (compare 2.2), in order to prevent them from interfering and distorting each other. Therefore multi-carrier transmission schemes usually do not offer a very high spectral efficiency as the subcarriers cannot be packed very closely together [4]. However OFDM, despite of being a multicarrier transmission scheme, offers a relatively high spectral efficiency [3]. This is achieved by introducing orthogonal subcarriers which will be explained in more detail later.

### 2.3.2 Physical Resource

Since in OFDM transmission the data stream gets transmitted over such a relatively large number of different carrier frequencies in parallel, it is common practice to illustrate the physical resource of the transmitted signal in both time and frequency domain, as shown in Figure 2.3.

In the horizontal axis, representing the time domain, we can see the OFDM symbols, each of which are transmitted during a time-duration $T_s$. In the vertical axis we can see the frequency band, containing the $N_C$ subcarriers of the OFDM signal. It is important to understand here that this picture only illustrates how the OFDM signal can be interpreted while it is transmitted over the radio channel but not how the LTE signal looks like during baseband processing. After the
receiver has executed steps like down-conversion, sampling and parallel to serial conversion, obviously this signal illustration is no longer valid. We will explain the complete transmission chain later in this chapter.

Figure 2.4 [3] illustrates a possible implementation for modulating an OFDM signal. This image shows what happens during one symbol duration time $T_s$.

### 2.3.3 Orthogonality

As mentioned before, multi carrier transmission schemes do not generally offer a high spectral efficiency due to the required guard bands between neighboring subcarriers. OFDM solves this problem by creating subcarriers which have got overlapping spectra and are pairwise orthogonal to each other. Due to their orthogonality, the subcarriers can be transmitted over carrier-frequencies which are very close to each other in frequency domain, but still do not interfere with each other. The orthogonality of the subcarriers is achieved by a rectangular pulse shaping and by choosing a suitable subcarrier spacing of $\Delta f = 1/T_s$ [4], where $T_s$ stands for the symbol duration time. The rectangular pulse shaping leads to a sinc-shaped spectrum for each subcarrier. In Figure 2.5 it can be seen how the sinc-shaped spectra of five OFDM subcarriers overlap with each other. A proof of the pairwise orthogonality of the LTE subcarriers can be found in [4].

In LTE the subcarrier spacing $\Delta f$ is specified as 15 kHz [17]. Depending on the total transmission BW, this means that the total number of subcarriers $N_c$
lays in the range between 75 and 1200.

### 2.3.4 Cyclic prefix

As mentioned previously, the subcarriers are pairwise orthogonal to each other due to the specific subcarrier spacing on the one hand and the sinc shaped spectrum of the subcarriers on the other hand. However, a time-dispersive radio channel can change the spectrum shape of a transmitted signal. If such a change in spectrum shape befalls the subcarriers of an OFDM signal, they will to some extent lose their pairwise orthogonality and since the spectra of the different subcarriers fundamentally overlap with each other, it is a reasonable assumption that this may result in strong interferences between the subcarriers. Another problem is that an inter-symbol interference might occur within single subcarriers due to multipath propagation. In order to make the OFDM signal robust against the corruption caused by time dispersive channels and multipath propagation, a so called cyclic prefix is added to the OFDM symbols in time domain. [3] This means that the last part of the OFDM symbol is copied and added again in front of the OFDM symbol.

### 2.3.5 Modulation using FFT processing

While Figure 2.4 illustrates the theoretical principle of creating an OFDM modulated signal, there is in common practice a faster method being used for OFDM modulation which includes FFT processing. In order to understand how FFT and IFFT processing can be used for OFDM modulation, we first recall that the formula for an inverse Discrete Fourier Transform $X$ of a sampled signal $x$ of length
N is defined as [18]:

$$X[n] = \sum_{k=0}^{N-1} x_k e^{j2\pi nk/N}$$

Let us now have another look at Figure 2.4 and recall that the difference between any two neighboring subcarrier-frequencies is always constant and defined as $\Delta f$. We can then conclude that it is possible to express an OFDM signal in baseband notation as:

$$X(t) = \sum_{k=0}^{N_c-1} a_k e^{j2\pi k\Delta ft}$$

When we talk about baseband notation here, we assume the OFDM signal has been down-converted, or not yet up-converted. In other words, if we modulate an OFDM signal by means of IFFT processing, we do not need several mixers for up-converting every single carrier frequency, but instead we can just up-convert the whole sum of subcarrier signals to a center frequency in the middle of the frequency band allocated by the carrier frequencies. Before up-conversion or after down-conversion the OFDM signal can be described as in (2.2).

In the following, we will consider a time-discrete version $X[n]$ of the signal, which can be obtained by sampling $X(t)$ with a sampling rate of $f_s$. We recall that in order to be able to fully reconstruct a time continuous signal from its time-discrete sampled form, the sampling theorem has to be fulfilled, which defines how high the sampling frequency $f_s$ has to be in comparison to the highest frequency component included in the signal. For the LTE standard it has been specified, that the sampling frequency $f_s$ should be a multiple of the subcarrier spacing $\Delta f$ and should be sufficiently higher than the transmission bandwidth $f_{tr}$ [3]. We can then express the sampling frequency as $f_s = N \times \Delta f$. 

**Figure 2.5.** OFDM subcarrier spectra
Furthermore, since $N_c$ is the total number of subcarriers we can easily confirm that the following inequality for the total transmission bandwidth $f_{tr}$ holds: $f_{tr} > N_c * \Delta f$. Using all these statements together we get:

$$f_s > f_{tr} \Rightarrow N * \Delta f > N_c * \Delta f \Rightarrow N > N_c$$  \hspace{1cm} (2.3)

We have thereby shown that the sampling frequency $f_s$ can be expressed by the term $N * \Delta f$ where $N$ must be larger than the total number of subcarriers $N_c$.

The following equation describes a discrete version of the OFDM baseband signal of (2.2), sampled with the sampling frequency $f_s = 1/T_s$.

$$X[n] = x(nT_s) = \sum_{k=0}^{N_c-1} a_k e^{j2\pi k \Delta f_s n / f_s} = \sum_{k=0}^{N_c-1} a_k e^{j2\pi k \frac{n}{N}}$$  \hspace{1cm} (2.4)

By comparing the last term of (2.4) with equation (2.1) we can see that this term is already very similar to and IDFT of our discrete input symbol sequence $a_k$ but that there is a difference in the upper border of the sum. Since $N$ has been specified to be larger than $N_c$ we can simply extend our discrete row $a_k$ with zeros to a length of $N$. In other words, we define a new discrete input signal $\hat{a}_k$ as follows: $\hat{a}_k = a_k$ for $k < Nc$ and $\hat{a}_k = 0$ for $N_c <= k < N$. By using this definition we can transfer (2.4) into:

$$X[n] = \sum_{k=0}^{N_c-1} a_k e^{j2\pi k \frac{n}{N}} = \sum_{k=0}^{N-1} \hat{a}_k e^{j2\pi k \frac{n}{N}} = \text{IFFT}(\hat{a}_k)$$  \hspace{1cm} (2.5)

**Figure 2.6.** OFDM modulation using IFFT

We have thus have shown, that the baseband representation of an OFDM modulated signal with $N_c$ subcarriers of the discrete input symbols $a_k$ can be
created by means of IFFT processing. Figure 2.6 [4] shows a schematic drawing of how the modulation is achieved. The demodulation can be achieved accordingly by using FFT processing as illustrated in Figure 2.7 [4].

![Figure 2.7. OFDM de-modulation using FFT](image_url)

We should note here that OFDM modulation and de-modulation by using FFTs and IFFTs are very attractive implementation methods due to their low computational cost [3].

### 2.3.6 Advantages and drawbacks

We can conclude that some of the main advantages of OFDM include its high spectral efficiency and a good robustness against frequency selective fading, due to the relatively narrow bandwidths of the subcarriers. Furthermore, it allows for highly efficient modulation methods by use of FFT/IFFT processing and for a straightforward realization of flexible transmission bandwidths, simply by adjusting the number of subcarriers which are allocated by the user of interest. Note that this flexibility was defined as one of the main LTE design targets in the specifications.

There are other advantages of OFDM that are beneficial from a mobile system perspective point of view. It offers for example a very good support for multi-broadcast transmission, which are transmissions that are sent simultaneously to all users in the system like for instance, mobile television [3]. Cell edge users who typically have a relatively poor performance can experience significant benefits for multi-broadcast reception in an OFDM system, because the received signals from different surrounding cells can be combined in their mobile [3]. The combination of signals from different cells usually introduces time dispersion due to the different distances between user and BS. However, in the case of OFDM, the cyclic prefixes can compensate for a big part of this time dispersion which may significantly
increase the signal quality [3]. There are more mobile system benefits of OFDM like its suitability as a user-multiplexing and multiple-access scheme which we will not discuss in detail since they are beyond the scope of this thesis.

One of the main drawbacks of OFDM is that it typically introduces large variations in the instantaneous transmission power [3]. This simply results from the fact that it is a multi carrier transmission scheme. We can understand this in a somewhat simplified way, if we just assume that introducing a larger number of subcarriers, will naturally cause the total transmission power to vary stronger over time. The problem with rapid and relatively strong changes in transmission power is, that they typically can produce nonlinearities in the transmitter which distort the signal [4]. In order to avoid or mitigate this issue, it is necessary to over-dimension the transmitter. But that consequently decreases the transmitter’s efficiency and increases power consumption. Thus OFDM is more suitable for the DL than for the UL, since in mobile terminals the power efficiency is much more crucial than in the base stations [4].

2.4 SC-FDMA

SC-FDMA (Single carrier frequency division multiple access) has been chosen as the transmission scheme for the LTE uplink. It is very similar to OFDM but the instantaneous transmission power variations are reduced by execution of an additional FFT processing step. For this reason SC-FDMA is also referred to as DFT-spread OFDM [4].

2.4.1 DFT-spread OFDM processing chain

One simple way to interprete the SC-FDMA transmission scheme is to see it as a conventional OFDM transmission, combined with a DFT based pre-coding [3]. Figure 2.8 shows some of the processing steps to be executed for SC-FDMA transmission. The receiver part of this chain has been implemented in Matlab and for the ePUMA. We should note here that some processing steps are missing in this figure such as, for instance, the channel equalization part. A more complete chain will be presented in chapter 3. However, we can observe that this processing chain looks very similar to conventional OFDM transmission except for one additional FFT processing step, namely an extra smaller length FFT processing block in the transmitter part and, analogously, one extra IFFT block in the receiver part. This extra FFT processing reduces the instantaneous power variations which makes SC-FDMA more suitable for uplink communication than OFDM [17]. Despite of this difference, SC-FDMA has got a lot of similarities with OFDM. So can, for example, the SC-FDMA physical resource as well be seen as a time-frequency grid with a subcarrier spacing of $\Delta f = 15$ kHz [10]. A difference is here though that, in case of SC-FDMA, the resources allocated to certain mobile users must consist of a set of consecutive subcarriers. The reason for this is that otherwise the single-carrier property of the uplink transmission would be lost. User multiplexing has not been considered in this thesis and thus this issue will not be explained any further here. For more details the reader is referred to [3] or [10].
2.5 Frame structure

The transmissions in the LTE standard are organized in so called radio frames with a length of 10 ms each. These radio frames are then further divided into 10 subframes which consist of 2 slots with a length of 0.5 ms each, as can be seen in Figure 2.9 [5].

One slot can consist of either 7 OFDM symbols, which is the typical case, or of 6 OFDM symbols in case an extended cyclic prefix is introduced [3]. Extended cyclic
prefixes are typically used if it is necessary to compensate for a high delay-spread introduced by the channel, or in case of multicast or broadcast transmissions [3], where the different time delays from different base stations to the user need to be compensated for. Figure 2.10 illustrates the time lengths of the cyclic prefixes in both normal and extended versions.

![Figure 2.10. Slot structure](image)

In this picture, $T_u$ represents the useful symbol time, and $T_{cp}$ and $T_{cp-e}$ stand for the normal and extended cyclic prefix time durations. In case a normal cyclic prefix length is used, the length of the first of the 7 OFDM symbols is with a total time duration of 5.1 $\mu$s a bit longer than the other six prefixes [3]. The reason for this is simply that all time durations in LTE are defined as multiple of a base unit $T_s$ and one slot with a time duration of 0.5 ms has got a number of 15360 of these time intervalls $T_s$ [3]. Since this number cannot be divided by seven, the first cyclic prefix has been specified to be slightly longer than the others.

The time frequency grid in LTE is furthermore divided into so called resource blocks and schedule blocks. One resource block consists of 12 subcarriers in the frequency domain and of one slot in the time domain. A schedule block consists of two resource blocks and thus has got the time duration of one subframe. As the name implies, a schedule block is the smallest unit of the physical resource grid that can be scheduled for a certain user. The reason for also defining the smaller resource blocks is, that LTE supports a so called frequency hopping transmission mode for which the transmission frequency of the user of interest changes on a slot basis [3].

In Figure 2.11 it is illustrated how the schedule and resource blocks are defined in the time frequency grid. The image shows four schedule blocks which are spread over two slots and a total bandwidth of 360 kHz. One resource block consists of up to $12 \times 7 = 84$ resource elements in total, which are the smallest physical units.
of the time frequency grid. One resource element carries the information of one complex modulated symbol.

One reason for organizing LTE in a radio frame structure including slots and subframes is related to LTE control signals as will be explained in the following. The radio signals, that are transmitted in the LTE standard, do not carry only the data which is of direct interest for the user, but they also contain many different types of control data, which ensures a sufficient service quality for the user. This control data includes for example signals for estimating and communicating the channel quality between users and base station in order to choose suitable modulation and coding schemes and to adapt the transmission power. Some of these control signals in LTE need to be transmitted more often than others and therefore certain control signals could for example allocate specific resources in just every radio frame and others in every single subframe.

In the next section we are going to discuss one specific control signal called demodulation reference signal which has been used for channel equalization in the baseband processing chain implemented in this thesis work.

2.6 Reference signals

Since radio channels vary over time and over different frequencies, it is necessary to estimate how the channel affects and corrupts the transmitted signal, in order
Reference signals

2.6 Reference signals

to successfully demodulate and recover the transmitted data at the receiver. We discussed earlier that in OFDM each subcarrier has got a relatively narrow bandwidth in frequency domain. Therefore, we can assume the corresponding channel of each subcarrier can be assumed to be non frequency selective, which means that its transfer function will not vary over different frequencies in this frequency band [14]. In the case of OFDM it is therefore not necessary to define a continuous channel transfer function \( H(f) \) over the whole transmission bandwidth at a certain point in time. We can instead describe the channel conditions in a discrete form by defining a complex value \( H_k \) for each of the \( n \) subcarriers. These complex values will naturally vary over time, as the channels of the subcarriers vary over time. In order to equalize the received signal, an estimate of these complex values is needed at the receiver side. A convenient solution for retrieving these channel estimates is to insert so called demodulation reference symbols, which are known at the receiver side, into the time frequency grid. The channel estimates \( H_k \) can then be calculated simply by dividing the received symbols \( y_k \) by the known transmitted reference symbols \( x_k \): \( H_k = y_k / x_k \). In LTE there are differences between the definitions of these reference signals for the up and downlink, as will be described in the following.

2.6.1 Downlink reference signals

The reference symbols in LTE are inserted into every single resource block in order to take the channel variations in time and frequency domains into account. In Figure 2.12 we can see an example of how the reference symbols are typically spread inside one schedule block for a downlink transmission [4]. The image illustrates only one out of six possible frequency shifts which are specified for the reference symbols [4]. Since we have got 12 subcarriers in total of which only two are allocated for the reference symbols, there are five more possibilities how the reference symbols can be divided in a schedule block. One benefit of this is that neighboring cells can use different frequency shifts for the reference signals, which reduces the interference of reference signals between cells [4].

Obviously the resource elements reserved for reference symbols cannot be used for user data transmission. The decision about how many reference symbols should be inserted into the time-frequency grid thus represents a tradeoff between prioritising data throughput or accuracy of the channel estimates. We should note here that it is furthermore possible in certain scenarios to improve the quality of the channel estimates, by averaging them over a certain number in time domain and/or in frequency domain. Whether such an averaging is beneficial, depends on the channel conditions more specifically its coherence time and coherence bandwidth. The coherence time of a channel is the maximum time period during which this channel can be assumed to be time invariant while coherence bandwidth is the maximum frequency band for which the channel can be considered to be non frequency selective [1]. Averaging over channel estimates of different time slots can therefore be beneficial if the coherence time of the channel is not too small, means that the channel conditions are not varying too fast over time. Analogously averaging over channel estimates from different resource blocks in frequency domain
would typically give better results in case of a high channel coherence bandwidth.

### 2.6.2 Uplink reference signals

In the uplink the reference signals are not defined for specific resource elements, but instead they are defined in certain DFTS-OFDM symbols and allocate the complete transmission frequency band as illustrated in Figure 2.13 [4]. As we can see, the reference signals are defined for every fourth OFDM symbol inside a slot so that every subframe contains two reference signal transmissions.

The main reason for defining complete DFTS-OFDM symbols as reference signals, is the previously discussed uplink requirement for low power variations during transmissions [4]. In order to keep power variations low, it is not suitable to frequency multiplex reference signal transmissions with other uplink data transmissions like it is the case for the downlink [4].

A reference signal consists of a certain number of complex reference symbols which are known to the receiver. Since the uplink reference signal is transmitted over the complete transmission bandwidth, this implies that the number of complex symbols of a reference signal sequence is equal to the total number of subcarriers included in this transmission bandwidth. Furthermore, since the transmission bandwidths are always defined over certain numbers of resource blocks with 12 subcarriers each, the number of complex symbols in a reference signal sequence
Reference signals

Figure 2.13. Uplink reference signals

will also be a multiple of 12 [4]. In general it is desirable for uplink reference signals to have the following properties [4]:

- The power should not vary too strong in frequency domain, in order to ensure relatively similar quality of channel estimation for different frequencies.

- The power variations in time domain should not be too strong either, since strong variations would decrease the power-amplifier efficiency.

- Neighboring cells should be able to use different reference signal sequences from each other, in order to reduce inter cell interference. This implies that it should be possible to define a sufficient number of different reference signals.

In order to achieve these targets in LTE, the uplink reference signals are defined to be extended versions of so-called Zadoff-Chu sequences, which have got constant power in both time and frequency domain and that makes them very suitable as reference signals[4]. A Zadoff-Chu sequence is expressed by the following equation[4]:

$$X_k^{ZC} = e^{-j \pi u(k(k+1)/M_{Zc})} \cdots 0 \leq k < M_{Zc}$$ (2.6)

The variable $M_{Zc}$ stands for the length of the Zadoff-Chu sequence. For a specific length $M_{Zc}$, not only one single Zadoff-Chu sequence exists, but rather
a set of different Zadoff-Chu sequences. This is indicated by the value $u$ in the equation, which is an index value that defines a specific sequence of the set of Zadoff-Chu sequences with length $M_{zc}$. The variable $u$ can take all integer values which are smaller than $M_{zc}$ and relatively prime to $M_{zc}$. Thus, we can conclude that if we define the length $M_{zc}$ of a set to be a prime number, there will be more Zadoff-Chu sequences available, compared to choosing a non-prime number in a similar order of magnitude as length. Since it is desirable to be able to define as many different reference signals as possible, Zadoff-Chu sequences with a prime-number length would be a very suitable choice as reference signals for LTE [3]. However, this is not possible because the sequence must be a multiple of 12 as we discussed earlier. Instead, the LTE standard defines the uplink reference signals to be cyclic extensions of prime-number Zadoff-Chu sequences [3]. This simply means that from a given transmission bandwidth with a certain number of subcarriers $n$, the largest prime-number smaller than $n$ is chosen as $M_{zc}$ [3]. Any of the Zadoff-Chu sequences of the set defined by $M_{zc}$ can then be transformed into a reference signal sequence with length $n$, by adding to it $n - M_{zc}$ cyclic extensions of the sequence.

Figure 2.14 shows an example, created with Matlab, of an LTE reference signal sequence of length $n=300$, defined by a Zadoff-Chu sequence of length $M_{zc} = 293$. We can see in the figure the constant power of the complex symbols, as they form a circle.

The reference signal length of 300, which has been used here, is equivalent to a transmission bandwidth of 5 MHz, as will be explained in more detail in the following chapter which describes the actual algorithms implemented in this thesis.
Chapter 3

Project implementation

In the following, we will present a high level description of the LTE uplink algorithms that have been implemented for the ePUMA processor. This chapter will provide the reader with an understanding of the project content and how the uplink chain was simulated in Matlab. The detailed ePUMA implementation will be described in later chapters.

3.1 Project overview

The purpose of this project work was to evaluate the potential of the ePUMA as a processing unit inside an LTE base station, which implies that it was of interest to either implement the downlink transmission part, or the uplink receiving part on the ePUMA. For this thesis, it has been decided to focus on the uplink receiving part. The complete chain for uplink baseband processing on the receiver side has been implemented first in Matlab and parts of it have then been mapped onto the ePUMA. However, since designing a complete functional LTE uplink receiver is a very challenging task, we had to make certain assumptions and simplifications in order to reduce the complexity to a suitable level. We can therefore describe the thesis work as an implementation of a highly simplified model for the physical layer processing of an LTE uplink receiver.

3.1.1 General assumptions

The following assumptions have been made for the transmitted signal which should be processed at the receiver side in this project.

- Total transmission bandwidth of 5 MHz
- No MIMO activated
- Frequency hopping deactivated
- One user transmits over the complete bandwidth
• 64 QAM modulation

The specific bandwidth of 5 MHz has mainly been chosen because it corresponds to a 300 point IFFT processing, which was useful to implement for the ePUMA because it would make implementation of other FFT sizes defined in LTE, a relatively easy task.

3.1.2 Transmission bandwidths

In line with the LTE design targets for a high spectrum flexibility, LTE supports a total of six different transmission bandwidths [4]. Since different transmission bandwidths correspond to different numbers of OFDM subcarriers, they also introduce different FFT lengths in the OFDM modulation and demodulation processing blocks. The following table shows the transmission bandwidths defined by the LTE specifications and their corresponding FFT/IFFT sizes. “Number of RB’s” describes how many resource blocks are included in the respective transmission bandwidth and the total number of subcarriers within the bandwidth is defined by the value “M”. We should recall here that the M-FFT sizes are only relevant for the uplink because they are a part of the SC-FDMA processing chain, but they are not included in the case of OFDM transmissions.

<table>
<thead>
<tr>
<th>Transmission BW</th>
<th>1.4 MHz</th>
<th>3 MHz</th>
<th>5 MHz</th>
<th>10 MHz</th>
<th>15 MHz</th>
<th>20 MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nr of RB’s</td>
<td>6</td>
<td>15</td>
<td>25</td>
<td>50</td>
<td>75</td>
<td>100</td>
</tr>
<tr>
<td>M-FFT size</td>
<td>72</td>
<td>180</td>
<td>300</td>
<td>600</td>
<td>900</td>
<td>1200</td>
</tr>
<tr>
<td>N-FFT size</td>
<td>128</td>
<td>256</td>
<td>512</td>
<td>1024</td>
<td>1536</td>
<td>2048</td>
</tr>
</tbody>
</table>

As we can see modulation/demodulation of an OFDM signal with a 5 MHz transmission bandwidth includes 512 point and 300 point FFT/IFFT processing steps. While a 512 point FFT algorithm had already been implemented for the ePUMA previous to this thesis work, implementing the 300 point FFT for the ePUMA was one of the main workloads of the project.

3.1.3 The uplink receiving processing chain

In the following section we will present all the processing steps that had to be executed by the ePUMA. Figure 3.1 illustrates the computational operations included in LTE UL layer 1 processing and highlights which of them were implemented for the ePUMA platform.

We can see in the illustration that as a first step the subcarrier spacing needs to be removed, which can be achieved by multiplying the input sequence with a sequence of appropriate phase shifts. This step is only necessary in the uplink processing chain, not in the downlink. Then, after cyclic prefix removal and 512 point FFT processing, the channel has to be estimated with help of the reference signal included in the fourth DFTS-OFDM symbol in every slot. By using the
3.2 Channel estimation and signal equalization

In the algorithm implementation for the channel estimation, we did not use any averaging between consecutive channel estimates over time or frequency domain. Instead, 300 complex channel estimates have been calculated for each slot, using the 300 reference symbols which are known at the receiver side and which are located in the slot’s forth OFDM symbol. These channel estimates have then
been used for equalizing the six remaining DFTS-OFDM symbols in the slot, which contain the data to be transmitted.

### 3.2.1 Channel estimation

If we assume that the coherence time of our channel is shorter than 0.5 ms, we can consider the channel to be time invariant for the time duration of one slot. It is therefore possible to model the channel for that time duration by a transfer function \( H(f) \) which is only dependent on the frequency. In frequency domain we then have the following relation between the signal \( X \) which is transmitted over the channel and the received signal \( Y \):

\[
Y(f) = H(f) \ast X(f)
\] (3.1)

However, as we mentioned before the subcarriers have got a narrow transmission-band which can be considered to be non frequency selective. In other words if the bandwidth of the subcarriers is smaller than the correlation bandwidth of the channel, the channel response will not vary over frequency within the band of one subcarrier. This is illustrated in Figure 3.2.

![Channel response](image)

**Figure 3.2.** Frequency selective channel transfer function

We can see that the channel response varies over different frequencies, but at the same time the channel response inside a subcarrier band is relatively constant. Thus, the narrow bandwidths of the subcarriers allow us to model the channel response of one subcarrier to be independent of the frequency domain. If we recall furthermore that each subcarrier carries exactly one complex symbol inside one DFTS-OFDM symbol we can define a vector \( X_k \) containing all complex symbols inside one DFTS-OFDM symbol, with \( 0 < k <= M \).

\( M \) represents here the total number of subcarriers included in the transmission bandwidth (300 in our case). From a baseband perspective, the channel transfer
function for each of the subcarriers can simply be represented by a complex value \( H_k \) which is multiplied with the complex symbol \( X_k \), carried by the subcarrier. If we then define the corresponding vector \( Y_k \), containing the received data symbols, we get the following representation:

\[
Y_k = H_k \ast X_k
\]  

(The channel estimates \( \hat{H}_k \) can therefore be calculated by:

\[
\hat{H}_k = \frac{Y_k}{X_k} = Y_k \ast \frac{1}{|X_k|} \ast \bar{X}_k
\]  

\[3.2\]

### 3.2.2 Channel equalization with Zero forcing

The channel equalization has been realized by implementing a zero forcing algorithm. Zero forcing is a straightforward method for equalization, where the received signal gets multiplied with the inverse of the channel response. The name zero forcing has been chosen, because this algorithm reduces the intersymbol interference to zero in case of noise free scenarios \[1\]. The main disadvantage of this equalization methods is, that filtering the received filter with the inverse channel response may amplify noise at specific frequencies where the channel spectrum has got a high attenuation \[1\]. For this reason zero forcing is typically not as often used for radio communication systems, as more robust methods like for example Least-Mean Square Equalization. However, for low noise scenarios the method performs well and can therefore be used for our simplified model in order to perfectly recover the data. The zero forcing equalization can be realized by implementing the following equation:

\[
\hat{X}_k = Y_k \ast \hat{H}_k^{-1} = Y_k \ast \frac{1}{|\hat{H}_k|} \ast \bar{\hat{H}}_k
\]  

(The expression \( \bar{\hat{H}}_k \) represents here the complex conjugate of the channel estimates. The reader might wonder here why the last term in \( 3.4 \) represents a simplification compared to \( \hat{X}_k = Y_k \ast \frac{1}{\hat{H}_k} \). As we will see in more detail later, for the ePUMA, being a fixed point processing platform, division is not a trivial task and it is therefore more convenient to implement a division with a real value, in this case represented by \( |\hat{H}_k| \), instead of dividing by the complex value \( \hat{H}_k \).

### 3.3 Implementation in Matlab

The receiver part of the previously described uplink processing chain has been implemented both in Matlab and for the ePUMA. Another software has been used to generate a digital DFTS-OFDM modulated LTE uplink signal with a transmission bandwidth of 5 MHz and 64 QAM as modulation scheme. We implemented then a simple channel model in Matlab in order to simulate the changes the LTE signal experiences, when being transmitted over a radio channel. Throughout the
Project implementation

complete project work matlab has been used as a reference and assisting tool for developing the different kernels or algorithms for the ePUMA.

![Diagram](image)

**Figure 3.3.** The project realization

The Matlab code for the uplink processing steps can be seen in the code example 3.1.

**Listing 3.1.** Matlab code LTE uplink processing chain

```matlab
1 1 % Import LTE signal
2 mydata= lte_fdd_UL_5MHz_64QAM_nofilter_basic (1:end,1)
3 +lte_fdd_UL_5MHz_64QAM_nofilter_basic (1:end,2)*1i;
4
5 1 % Create channel response
6 hcoef=randn(3,1)+randn(3,1)*1i;
7 h = [hcoef(1)*3 0 0 2*hcoef(2) 0 hcoef(3)];
8 h = h/norm(h);
9
10 1 % Create received signal by convoluting transmitted
11 1 % signal with channel response
12 rx_signal=conv(mydata.', h').';
13 rx_signal=rx_signal(1:end-5);
14
15 1 % Cyclic prefix removal
16 sym(:,1)=rx_signal(41:(41+511));
17 offset =40+512;
18 sym16=rx_signal( offset+1:( offset+(512+36)*6));
19 sym16=reshape(sym16, 512+36,[]);
20
21 1 % Subcarrier spacing removal
22 sym(:,2:7)=sym16(37:end,:);
```
% 512 point FFT processing step
rx_sym_fft = 1 / sqrt(512) / sqrt(512 / 300) * fft(halfc_shifted_sym);
rx_shift = circshift(rx_sym_fft, 150);

% Compute channel estimates
hest = reference_signal_mul.*rx_shift(1:300,4);
conj(hest) = c_h;

% Channel equalization
rx_shift(1:300,1) = 1./(hest.*c_h).c_h.*rx_shift(1:300,1);
rx_shift(1:300,2) = 1./(hest.*c_h).c_h.*rx_shift(1:300,2);
rx_shift(1:300,3) = 1./(hest.*c_h).c_h.*rx_shift(1:300,3);
rx_shift(1:300,5) = 1./(hest.*c_h).c_h.*rx_shift(1:300,5);
rx_shift(1:300,6) = 1./(hest.*c_h).c_h.*rx_shift(1:300,6);
rx_shift(1:300,7) = 1./(hest.*c_h).c_h.*rx_shift(1:300,7);

% 300 point IFFT processing step
rx_sym = sqrt(300) * ifft(rx_shift(1:300,:));
scatterplot(reshape(rx_sym(:,1:3),[],1));

Executing the code above will generate the following figure, in which we can see the 64QAM constellation, which indicates, that the signal has succesfully been demodulated.

Figure 3.4. 64QAM constellation after signal demodulation
Chapter 4

The ePuma DSP platform

In the following chapter we will give an introduction to the ePUMA architecture and its processing components. The main scope of this chapter is to provide the reader with the necessary background for understanding the implementations that have been made in this thesis, rather than including all details of the ePUMA architecture. For a more complete introduction to the ePUMA the reader is referred to [12] and [21].

4.1 Background

Recent trends in the development of computer technology show an increasing importance of parallel computing methods to fulfill the ever rising demands on computational performance. Parallel computing methods can be distinguished into the following three different categories.

1. The development and programming of multi-core processors which is called task parallelism.

2. The usage of processors that operate on vectorized data which utilizes the so called data parallelism. An example for this kind of processors operating on vectorized data are SIMD’s, which stands for “Single instruction multiple data” [15].

3. A third form of parallelism finally, that can be exploited by parallel computing methods is the instruction level parallelism, where the general idea is to execute different instructions simultaneously, if their results do not depend on each other. One technique that makes use of instruction level parallelism is the pipelining approach, where the execution times of consecutive instructions partially overlap with each other.

Several reasons for the increased usage of parallel computing methods in general and the development towards multi-core processors in particular can be found.
Many of them are related to the following three factors, which cause significant limitations for further performance improvements of traditional single core processors [13]:

- Performance improvements reached by using higher CPU clock frequencies have basically come to an end mainly because this method causes unsatisfactory high levels of power consumption.

- The instruction level parallelism has already been exploited to large degrees and thus in general little further performance gains are to be expected from this method.

- Memory speeds so far cannot keep up with the ever increasing processor speeds, which limits the overall performance improvements reached with higher processor speeds.

The trend towards higher computing parallelism can be observed both for general purpose computers as well as for embedded systems which typically target real-time signal processing. The development of the highly parallel ePUMA DSP platform is therefore coherent with the current developments in computing technology.

The ePUMA is planned to be used in various DSP fields like for example baseband signal and radar signal processing, video games and video coding and decoding [12]. The main design goal is to offer an embedded platform capable of high performance parallel computing while simultaneously consuming little power and having low silicon cost [21]. In order to achieve that purpose ePUMA has been designed in an innovative way that enables the platform to exploit different forms of computing parallelism. The design combines the multi-core approach with the SIMD approach and furthermore includes a unique memory subsystem which is aimed at hiding memory access time behind the computing time of the program [21],[12].

### 4.2 The ePUMA architecture

In the following we will give a brief overview of the ePUMA architecture. The platform contains a master RISC (Reduced Instruction Set Computing) DSP processor and eight SIMD co-processors together on a single chip [12]. This multi-core design including eight processors with vectorized data processing capabilities is the key to epumas potential for high performance computations since it allows for strong utilization of task- and data-parallelism simultaneously. Signal processing tasks, that can be vectorized, can be divided onto up to eight SIMD processors, which are all capable of high throughput for arithmetic calculations. The SIMD cores are therefore intended to be used as the main data processing units of the ePUMA platform. The master DSP processor on the other hand, has got various tasks including:

- Control of the overall ePUMA program flow and execution which includes for example the transmission of start commands to the SIMDs.
4.3 Memory subsystem and data communication

- Control of the data transfer between main memory and SIMDs by configuration and initialization of DMA transactions.
- Calculating parts of application algorithms which cannot be vectorized and therefore not efficiently handled by the SIMD processors.
- Computation of smaller tasks for which transferring them to the SIMDs would mean waste of computational resources.

A schematic overview of the ePUMA architecture can be seen in Figure 4.1 with the master processor in the center, which is surrounded by the eight SIMD co-processors. The switching nodes N1 - N8 are part of the on-chip network which enables communication between master and SIMD processors. The communication from master (or main memory) to the SIMD processors or vice versa is executed over the so called “star-network”, named according to the shape of the overall ePUMA chip design. We can also see the ring network, which is highlighted by a thicker line in Figure 4.1 and which connects the SIMD processors with each other. For data access on the main memory a DMA controller is used.

**Figure 4.1.** ePUMA architecture. The picture is inspired by Figure 1 in [21]

4.3 Memory subsystem and data communication

In this section we will give a more detailed description of the ePUMA memory subsystem and memory hierarchy and how the data is communicated between different memory components over the on-chip network.
4.3.1 Memory hierarchy

The complete ePUMA architecture contains various different types of memory. For a better overview it is convenient to divide the different memory components into three different groups according to the accessibility of the data stored in them. In [6] the different groups are named level 1-3, while level 1 represents the memory with highest accessibility by the master or SIMD co-processors. We will refer to the memory groups by these titles in the following. In Figure 4.2 the different memory levels are illustrated [6].

![Figure 4.2. ePUMA architecture](image)

As we can see in the figure, level 3 memory is not located on the chip but an external memory. It is the biggest sized memory and contains all program and computing data [6]. Its access is handled by a DMA-controller, which is configured and controlled by the master processor.

Both level 1 and level 2 memory components on the other hand are located on-chip and can thus directly be accessed by the processor. We can observe that for both processor types (master and SIMDs) the level 1 memory consists exclusively of a vector register file (VRF). The level 2 memory group contains program memory (PM) and additional memory components which differ for master and SIMD processors. In case of the master processor these components consist of two data memories (DM0/DM1) while each of the eight SIMDs contains one constant memory (CM) and three local vector memories (LVM1-LVM3).

Despite of the fact that both level 1 and level 2 memory components can be directly accessed by the processors there are still differences in the accessibility between the two groups. The level 1 memory offers the highest data accessibility
because the registers contained in the vector register file have got less limitations with respect to how often they can be accessed per clock cycle. It would for example be possible for an unlimited amount of time to continuously execute an instruction that adds two registers of the VRF together and writes the result into a third register. This means the VRF is accessed three times per clock cycle namely for reading the two operands and for storing one result. In case of the LVMs in level 2 this would not be possible, because this memory type can only be accessed twice per clock cycle, once for reading and once for writing to it.

4.3.2 Data communication

In order to start the data processing on the ePUMA platform both input data as well as the programs running on the SIMDs have to be moved from the external main memory to the level 2 local memory components of the ePUMA chip. At the beginning of any ePUMA data processing, the master processor would typically initialize a data transfer of the SIMD programs (or kernels) into the program memories (PM) of the SIMD co-processors which shall be involved in the data processing. This communication of program data is configured exclusively by the master program. That is also the case for initializing the constant memory (CM) of the SIMDs via data transfer from the main memory.

After the kernels of the SIMDs have been successfully loaded a typical next step could be to load input data from main memory to the local vector memories (LVMs) of the SIMDs. This type of communication now requires, except for configuration in the master software, additionally a configuration in the SIMD kernel, which then controls a packet processing unit (PPU). Each of the eight SIMDs has got such a PPU which is involved in any communication of input-data from main memory to the LVMs or of output data which shall be communicated from LVMs to main memory. The different types of star-network communication that are possible on the ePUMA are illustrated in Figure 4.3.

As we can see the PPU is involved in every communication between main memory and the level 2 memory components of the SIMD co-processors. However only data transfer with the local vector memories (LVMs) needs PPU configuration by the SIMD software. The reason for this is simply, that the loading of program memory and constant memory is done before the SIMD of interest has started any processing. The SIMDs will be initialized in an IDLE state, where they merely wait for program- and constant data which does not require any further configuration of the PPU. Any data exchange with the LVMs on the other hand will take place during SIMD program execution and thus it is necessary to coordinate at which point in the SIMD program the data communication shall take place. Furthermore it is possible to specify inside the SIMD software the exact addresses inside the LVMs, were the PPU shall store or read the data. Similarly the master software specifies the addresses in the main memory where the data that shall be exchanged is located.

The ring network communication, means the data transfer between LVMs of different SIMDs, is executed by the two PPUs of data transmitting and data receiving SIMDs. Naturally this type of communication requires configuration in the
The ePuma DSP platform

Figure 4.3. Star network communication

programs of both SIMDs participating in the data transfer. Since ring network communication does not include level 3 memory no DMA controller configuration is required. However, the master processor is nevertheless involved in this communication since it is responsible for setting up the switching nodes on the on-chip network in order to create the physical connection between the two SIMDs of interest. If two SIMDs are configured to exchange data with each other, but no physical connection between them exists, they will interrupt the master processor, and as part of the interrupt routine, the master will configure the necessary link on the on-chip network [6].

4.4 The ePUMA Master

The master processor of the ePUMA is commonly referred to by the name “senior” or “senior processor” by the ePUMA research team. It is a so called single issue DSP processor, which means that exactly one instruction can be announced in each clock cycle [12]. The processors intended usage areas include many different DSP applications like for example voice and audio coding and decoding, bit manipulations and program flow control for video coding and decoding [7]. As mentioned earlier, when used as a master processor for ePUMA, the senior processor will have to handle various tasks including SIMD control, coordination of data transfers via DMA controller and on-chip network and execution of smaller computational tasks. Due to this relatively big variety of tasks it is assumed that the senior processor might become a bottle-neck of the ePUMA platform in certain situations and therefore a future replacement by another processor with higher computational performance is possible [6]. At the time of the thesis work the se-
The SIMD processors

The eight SIMD-co processors are the main data processing units of the ePUMA platform and are as well referred to by the name sleipnir processor. Each sleipnir is capable of arithmetic operations on vectors with a length of 128 bits which is the key contributor to the high parallel computation power of the ePUMA. In this thesis work kernels for 300 point FFT computation as well as for a zero forcing channel estimator have been implemented for the SIMD processors. Therefore we will in the following give some introduction to the sleipnirs architecture, datapath, pipeline and instruction set to provide the reader with the necessary background knowledge for understanding the implementations made in this thesis.

4.5.1 Supported datatypes

The sleipnir is as mentioned earlier a fixed point processor that is able to compute arithmetic operations on data vectors with a total size of 128 bits. (For an in-depth explanation of fixed-point numerical representation the reader is referred to [15].) The number of data values which are included in one 128 bit vector can vary and depends on the format of the data which shall be processed. The sleipnir processor is capable of interpreting the 128 bits in different ways by supporting the following data formats which are listed in order of increasing lengths of the data values[9]:

- **Bytes**: Represent data values with a length of 8 bits.
- **Words**: Data values with a length of 16 bits.
- **Double words**: Data values with a length of 32 bits.
- **Complex words**: Complex data values with a length of 32 bits, where the first 16 bits represent the real part and the last 16 bits the imaginary part of the complex data.
- **Complex double words**: Complex data value with a total length of 64 bits and 32 bit real and imaginary parts.

All of these data types can furthermore be interpreted as signed or unsigned data values by the sleipnir processor. In Figure 4.4 all the data formats which are
supported for the 128 bit data vectors are illustrated. We can see that one vector can contain between 2 and 16 data values depending on their format.

The SIMDs ability to handle all these different types of data is achieved by introducing and adding different instructions for different data formats to the processors instruction set. There are for example different instructions defined for multiplication of different data types. We should furthermore mention that the sleipnir can also operate on data vectors of smaller size than 128 bits. It is for example possible for the processor to execute certain instructions on half vectors (64 bits), double words (32 bits) and words (16 bits). In the following sections we will discuss the hardware components included on the sleipnir more specifically its datapath.

4.5.2 Sleipnir datapath

The data path is the part inside a processor were the actual data processing is executed. It typically includes various hardware components like multipliers or arithmetic logic units (ALU) for data manipulation. Operands (input data values) are moved from processor registers or memories to the datapath, were they are processed and afterwards the results are saved back to registers or memories [15]. Based on the instructions issued inside the processor, the hardware components inside the data path will be connected by switches in specific ways, so that different computational operations correspond to different instructions. In modern DSP processors, the datapath is usually of a pipelined structure, which means, that an instruction will take more than one clock cycle until its execution is completed. The processing of consecutive instructions will partially overlap with each other.
as they move step-wise through the different stages of the data path [15].

For the sleipnir co-processor, the data path consists of three stages of processing hardware elements. The first one is the multiplying stage, which consists of sixteen 16-bit multipliers and which is followed by two ALU stages [12]. In order to enable the sleipnir of data processing for all the different data-formats mentioned earlier, the 128 bits supported by the data path are split up into eight 16-bit paths (usually referred to as 16-bit lanes) before entering the data path.

Execution time of instructions can vary significantly in the sleipnir processor due to their different ways through the datapath and their different numbers of pipeline stages. The shortest instructions only have got two pipeline stages while more complex instructions consist of up to thirteen different pipeline stages. This needs to be taken into account when writing kernels for the sleipnir processor as will be discussed in more detail later.

4.5.3 Memory components and registers

In this section we will give an overview of all the sleipnir’s memory components while explaining their functionality and mention some of their technical details. These components include [9]:

- **Vector register file (VRF)** Main register file of the processor with the highest data accessibility. Can save up to eight vectors of 128 bit.

- **Local vector memory (LVM)** The main storage unit for the data being processed. Each sleipnir has got access to three LVMs with a storage capacity of 4096 128-bit vectors each.

- **Program memory (PM)** Saves the software of the sleipnir and has got a total saving capacity for 1024 instructions in the program.

- **Constant memory (CM)** A memory for constant data values and permutation and access patterns, which are used for addressing LVMs or VRF. It can save up to 128 vectors of 128-bit length.

Furthermore does the sleipnir contain eight 40-bit accumulator register, a vector flag register which saves the eight flags caused by the operations in the eight datapath lanes, and a number of so called special purpose registers. These registers fulfill particular tasks by saving data that enables specific operations like for example addressing the LVMs or the constant memory. In the following section we will explain some limitations about the accessibility of the data in the memory components.

4.5.4 Data access of LVMs and vector register file

While the LVMs represent the main storage for the data to be processed, the VRF typically would save intermediate data due to its highly flexible data accessibility. There are, however, some limitations for how data can be accessed from these two memory components. Both vector register file and LVMs are each organized in
eight so-called memory banks with sizes of 16 bits. If one specific data vector is fetched from one of these memory components (LVM or VRF) each of the eight memory banks can be accessed only one time. Similarly if a 128-bit data vector is written into the VRF or one of the three LVMs, each of their eight memory banks is available for receiving exactly one 16-bit value. Figure 4.5 shows an illustration of the VRF which is capable of storing eight 128-bit vectors. Each of the numbers from 1 to 64 represents a 16-bit data value. The rows represent the eight vector registers and the columns represent the eight memory banks. The only limitations for data access which follows from the data organization in memory banks is, that any two values which are saved in the same memory bank can never be accessed simultaneously like for example data value 1 and data value 9. Two examples for data-value vectors that can be fetched from the vector register file in this case are the data sets 1,2,...,8 or 9,26,35,20,53,62,47,48 as illustrated in Figure 4.5.

![Figure 4.5. Memory banks](image)

The LVMs are organized exactly in the same way with the difference, that each memory bank is not only able to store a number of eight 16-bit values as the VRF but 4096 instead. When we consider that for each of the eight memory banks any single data value of these 4096 potentially stored values can be fetched for a 128 bit operand at any time, we can conclude that despite of this afore mentioned limitation the LVMs still offer quite a high flexibility with respect to their overall data accessibility. Let us now return to our example illustrated in Figure 4.5 and assume that the 64 16-bit values represent a Matrix of size 8 * 8. If we were for example interested in calculating both the sum of the row vectors as well as the sum of the column vectors of this matrix by using SIMD vector operations, we would realize that this would not be possible with the current data arrangement, because the data inside the column vectors cannot be accessed simultaneously.
The solution to this problem is to permute the data as illustrated in Figure 4.6.

![Figure 4.6. Data permutation in the VRF](image)

We can see that the circular shifting of the data values contained in the row vectors allows for data access of both row and column vectors. It is illustrated how the data values which were located in memory banks 1 and 5 in Figure 4.5, can now be fetched into a single data vector simultaneously due to the fact that they are now located in different memory banks. This general concept of data permutation for better accessibility is a technique that typically is used a lot when implementing kernels for SIMD processors and has also been used in the implementations for the sleipnir processor in this thesis. After the eight data values have been fetched from the eight memory banks, it is furthermore possible to permute their exact position inside the 128-bit vector in any desired way. An example of this method is illustrated in Figure 4.6 for the data values previously located in memory bank 5.

### 4.5.5 Programming the sleipnir

As mentioned earlier the sleipnir processor is programmable in assembly language. Higher level language would currently not be able to efficiently exploit the parallel computing potential of the processor due to the limitations of the latest compiler technology and even in the future the usage of a higher level language for programming the sleipnir is considered to be relatively unlikely[12]. In the following we will provide a brief introduction to the assembly programming of the sleipnir with focus on instructions that have been used in this thesis work.
Addressing methods

In this section we will explain how the different memory components and their addresses can be referred to by the sleipnir software. In order to do so we need to first give a brief introduction to the assembly instructions, which have got the following basic structure:

- operation destination src1 src2

“Operation” represents the instruction to be executed, which could for example be an arithmetic calculation like an addition. “Destination” defines were the result shall be saved and “src1” and “scr2” are the input operands on which the operation shall be executed. Source operands and destinations are defined as specific addresses in either VRF, LVMs or the constant memory with the limitation that the CM cannot be specified as destination. We see from the basic structure, that a sleipnir assembly instruction includes up to three addresses. (Obviously less addresses may be required for certain instructions like for example a copy command.) In the following we will explain the different methods for specifying these addresses, which are supported by the sleipnir assembly language. Table 4.1 lists how the different memory types can be referred to by the sleipnir assembly code.

<table>
<thead>
<tr>
<th>Memory component</th>
<th>Assembly reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector Register file (VRF)</td>
<td>“vr0” - “vr7” or “vrf[address]”</td>
</tr>
<tr>
<td>Local vector memory (LVM)</td>
<td>“m0[address]” or “m1[address]”</td>
</tr>
<tr>
<td>Constant memory (CM)</td>
<td>“cm[address]”</td>
</tr>
</tbody>
</table>

We notice that there are two ways of addressing data vectors inside the VRF. If the expressions “vr0” to “vr7” are used, one of the eight 128-bit vectors, which are illustrated as rows in Figure 4.6, will be the address that the program refers to. If the reference “vrf” is used instead, more flexible addressing methods are possible including permutation of data values and accessing any of the eight addresses inside each memory bank.

Two of the sleipnir’s local vector memories can be addressed as ”m0“ and ”m1“, and the addressing offers the same flexibility as the addressing of the vector register file by the name ”vrf“ with regard to permutation and memory bank access. The reader might wonder here, why only two of the three LVMs can be referred to by the assembly code. The reason for this is, that one of the LVMs will always be available for data exchange with the ring or the star network while only the other two will be used for data processing. This way it is possible to hide memory access time behind program execution time. Which of the three LVMs shall be used for data exchange and which of the other two data memories will be referred to as m0 and m1 can be configured and changed at any time during SIMD execution via switches, which are controlled by the sleipnir program.

The constant memory finally is always referred to by ”cm“ and can be accessed by specifying an address to any of the up to 128 data vectors (of length 128-bits
each) which it can store. If a vector is accessed the CM does not support any permutation of the 16-bit data values that are contained in this vector. However the access of smaller data-sizes than 128-bits, like for example half-vectors, is possible.

For a better understanding of the sleipnir assembly language let us consider the following example code 4.1 which includes some vector addition and multiplication instructions. We should mention here that in all the following code examples any text in a line behind "//" signs merely represents comments that are added to provide a better understanding for the reader and do not belong to the sleipnir code.

**Listing 4.1. Sleipnir example code**

```
addw vr2 vr0 vr1
addw vrf[32].v vrf[24].v vrf[0].v // = addw vr4 vr3 vr0
cmulwww m0[0].v vr2 ml[0].v
cmulwww m0[8].v vr2 cm[0].v
```

The instruction addw stands for addition of 16-bit data words. The first line adds the eight 16-bit data values located in register vr0 together with the eight values stored in vr1 and writes the eight 16-bit results into register vr2. The second line performs the same operation but for different registers (see comment) and the alternative syntax for vrf addressing is used.

The command cmulwww defines a complex multiplication. The three w’s specify, that both operands and the destination-vector contain 32-bit complex words as data values. Thus each of these instructions executes four complex multiplications of 32-bit complex data words. Furthermore we see that LVMs and CM are involved as addresses for operands and destinations. We should note here that contrary to the cmulwww instruction, the addw instruction does not specify if the 16-bit words represent complex data or real data. That is not necessary since there is no computational difference between the addition of eight 16-bit real words or four 32-bit complex words.

The notation .v in the code specifies, that a complete vector shall be accessed. This is not necessary when using the references "vr0" to "vr7", since these expressions already define by themselves complete vector registers. Access of shorter data lengths can be achieved as illustrated in Table 4.2.

**Table 4.2. Addressing of different vector lengths**

<table>
<thead>
<tr>
<th>Type</th>
<th>Length</th>
<th>VRF syntax 1</th>
<th>VRF syntax 2</th>
<th>LVM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector</td>
<td>128 bit</td>
<td>vr0</td>
<td>vrf[addr].v</td>
<td>m0[addr].v</td>
</tr>
<tr>
<td>Half Vector</td>
<td>64 bit</td>
<td>vr0.Yh</td>
<td>vrf[addr].h</td>
<td>m0[addr].h</td>
</tr>
<tr>
<td>Double Word</td>
<td>32 bit</td>
<td>vr0.Yd</td>
<td>vrf[addr].d</td>
<td>m0[addr].d</td>
</tr>
<tr>
<td>Word</td>
<td>16 bit</td>
<td>vr0.Y</td>
<td>vrf[addr].w</td>
<td>m0[addr].w</td>
</tr>
</tbody>
</table>
For better readability we merely illustrated examples for vr0 and m0. The Y specifies from which starting position inside the complete vector, the data of shorter length shall be fetched from and thus its range of values depends on the bit size of the data values that shall be accessed. In case of the data type word for example, Y could be defined as any integer value between 0 and 7, representing the eight 16-bit values included in the complete vector. It is furthermore possible for certain instructions to define different lengths for operands and destinations as shown in listing 4.2.

**Listing 4.2. Different operand and destination data types**

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>addw vr2</td>
<td>vr0</td>
<td>vr1. 7w</td>
</tr>
<tr>
<td>2</td>
<td>tcmacw</td>
<td>vr3. 0d</td>
<td>vr1</td>
</tr>
</tbody>
</table>

We can see that for the addw instruction one of the two operands merely consists of a 16-bit data word, while the other operand and the destination are addressed as complete vectors. This instruction will simply add the data word stored in the end of register vr1 to each of the eight data words in vr0. The eight 16-bit results of these eight additions will then be written to vr2.

The instruction tcmacw performs four complex multiplications of complex words and adds the four results together. Therefore the output of this instruction consists only of one 32-bit complex word and the destination needs to be addressed as a double word. The tcmacw instructions has been an important instruction for implementing the 300 point FFT kernel for the sleipnir processor in this thesis.

So far we only discussed addressing the LVMs and VRF by usage of immediate values. If we for example address m0[0].v the first eight 16 bit values inside the LVM will be accessed, means LVM addresses 0 to 7. However the address generation for the LVM is actually more complex and does not necessarily consist of only one immediate value. The address gets generated by two parts, one scalar part for the start address and one pattern part, which is a vector that specifies the permutation pattern for the eight data values to be accessed [9]. The scalar part can be further divided into a sum of an index value and an offset value. We can then write the generated vector address as [9]:

$$address[i] = \text{index} + \text{offset} + \text{pattern}[i], i \in [0, 1...7]$$

The offset part is a 16 bit scalar value, that can either be an immediate value or can be fetched from the vector register file [9]. The index part is fetched from one of the special purpose registers used for addressing [9]. If index and/or offset values are not defined in the instruction they will be set to zero by default. If the pattern instruction is not specified, it will be defined according to Table 4.3 [9].

If we look at these default permutation patterns we can understand why addressing m0[0].v will fetch the first eight consecutive 16-bit addresses in the LVM and why the addition of data types of different sizes will work the way we stated earlier for our code example 4.2. In the following we will take a closer look at how the index part of the address is provided by the special purpose registers.
Table 4.3. Default permutation patterns

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Type</th>
<th>Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>v</td>
<td>Vector</td>
<td>0,1,2,3,4,5,6,7</td>
</tr>
<tr>
<td>h</td>
<td>Half Vector</td>
<td>0,1,2,3,0,1,2,3</td>
</tr>
<tr>
<td>d</td>
<td>Double Word</td>
<td>0,1,0,1,0,1,0,1</td>
</tr>
<tr>
<td>w</td>
<td>Word</td>
<td>0,0,0,0,0,0,0,0</td>
</tr>
</tbody>
</table>

As mentioned earlier, the sleipnir processor design includes a number of registers that are exclusively reserved for addressing purposes. In total the processor includes ten of these registers and each of them is used for addressing one specific memory component of VRF, LVMs or constant memory. A complete overview of the addressing registers can be seen in Table 4.4.

Table 4.4. Special purpose registers for addressing

<table>
<thead>
<tr>
<th>Component</th>
<th>Nr of registers</th>
<th>Register name</th>
</tr>
</thead>
<tbody>
<tr>
<td>CM</td>
<td>2</td>
<td>car0, car1</td>
</tr>
<tr>
<td>LVM</td>
<td>4</td>
<td>ar0-ar3</td>
</tr>
<tr>
<td>VRF</td>
<td>4</td>
<td>rar0-rar3</td>
</tr>
</tbody>
</table>

The main purpose of these addressing registers is to reduce the sleipnir code size, because they support automatic incrementing of their values between consecutive instructions, which will decrease the number of necessary code lines significantly. There are a number of differences between the address registers for different memory components and their usage, which will be discussed in the following.

The constant memory is only vector addressable. The expression cm[0] refers for example to the first 128-bit vector and cm[1] to the second vector stored in CM. The address registers car0 and car1 therefore support incrementing or decrementing of the value 1, which is then equivalent to accessing the next or previous vector inside the constant memory.

The LVMs on the other hand can address any 16-bit value in any memory bank as their start address and thus the addressing is not defined on a 128-bit vector basis but for 16-bit data words instead. When addressing the LVM, the expression m0[0].v will refer to the first vector stored in the LVM with the eight 16-bit addresses 0 to 7. However m0[1].v will refer to addresses 1 to 8. If the second vector of the LVM shall be accessed, means addresses 8 to 16 we would have to use the reference m0[8].v. Incrementing the registers ar0-ar3 (referred to as arX in the following) with the value 1 will therefore not access the next vector but instead move the address pointer one 16-bit address further. Moving the pointer to the consecutive 128-bit vector could be achieved by incrementing with the value 8. The arX registers thus support incrementing or decrementing its pointer with the values 1,2,4 and 8. Furthermore can each of the arX registers store a step value, which can be configured during program execution with any desired value and which can as well be used for incrementing the arX pointers.
Both arX and carX registers furthermore support so called cyclic addressing, by storing a top and a bottom address. If the the carX or arX register gets incremented while pointing at their top address, the pointer will jump back to the bottom address, if cyclic addressing was activated. Cyclic addressing can be indicated in the sleipnir code by using a % after the incrementing operand. (For example ar0+=1% instead of ar0+=1)

The VRF supports cyclic addressing as well, but the bottom register is always the first address in the VRF (address 0) and the top register is the last address included in the VRF(address 63) and cannot be specified otherwise. Another difference for the VRF is, that the values of its address registers rarX can be set immediately and not only by copy commands like it is the case for arX and carX registers. For better understanding how the addressing with these special purpose registers helps reducing the code size let us consider the following sleipnir code example 4.3.

Listing 4.3. Adressing methods with special purpose registers

```sleipnir
// initialising LVM address registers
1 copy ar0 cm[AR0_init]  //cm[0]
2 copy ar1 cm[AR1_init]  //cm[1]
3 // no new instructions until copy commands are finished
4 7*nop
5 // Issues eight vector additions
6 8*addw m0[ar0+=8].v ml[ar1+=s%].v vrf[rar1=0, rar1+=8].v
7 .cm // Constant memory section
8 AR0_init
9 // start bottom top step
10 23 0 0 0 0 0 0
11 AR1_init
12 0 0 63 16 0 0 0 0
```

Firstly, we should note that this example program consists of two sections which are separated by the "cm" in number nine. Everything above this line is the program code while everything below represents the data that the constant memory of the sleipnir processor shall be initialized with. As we can see the constant memory stores in our example only two 128-bit vectors with eight 16-bit data-values each. The expressions "AR0_init" and "AR1_init" are strings that can be used to reference these two addresses inside the constant memory. The instructions "cm[0]" (first vector in cm) and "cm[AR0_init]" would thus be identical references pointing to the same data vector inside the CM.

In the code section in line 2 and 3 the special purpose registers for LVM addressing ar0 and ar1 get initialized with the two vectors stored in CM. The comment in line 11 illustrates the structure of how these registers are initialized. The first four 16-bit data values include the start address, top and bottom addresses for cyclic addressing, and the step value for incrementing the address value. The last four
values of the vectors are not used for the special registers. Before ar0 and ar1 can be used, seven so called “nop” instructions are executed. The command “nop” stands for “no operation” and is needed due to the pipelined architecture of the sleipnir. Since consecutive instructions partially overlap with each other these nop commands are needed here in order to ensure that the initialization of the registers has been finished before they are used for addressing. In line number seven we can see how the addressing methods of the special purpose registers can be used to reduce the sleipnir code size. This code line represents eight vector-additions with different addresses for destination and source operands. The destination address is located in the LVM referred to as “m0” and the first vector-result will be stored starting at address 23, the value which ar0 has been initialized with. After each addition the ar0 pointer will be incremented by eight so in this example the eight results from the vector-additions will be stored in the LVM without any gaps in between them. The two operands addresses consist of another LVM referred to by m1 and the VRF. We can see that the VRF address is as well incremented by the value eight after each addition and that the start address for its used special register rar1 is initialized to zero by an immediate value. For the LVM operands in m1 we see that the address is incremented between additions with the step value “s” which has been initialized to 16 in this case. Furthermore, cyclic addressing is activated (indicated by the %) which means that after four additions when the top value 63 is reached, the address pointer will jump back to the bottom address 0.

Sleipnir instruction structure

As we saw in the previous code example, the sleipnir language allows the repetition of instructions simply by using a scalar value in front of the code line. The complete structure of sleipnir instructions is as we could see a bit more complex as earlier described and looks as follows [8]:

- \[\text{N* operation.cdt <options> destination scr1 src2}\]

The new parts here are the previously mentioned “N” which simply defines the number of times that the code-line shall be executed. The “.cdt” stands for “condition” and can for example be used to execute the instruction only under certain conditions. Since conditions have not been used in this project, we will not explain them in more detail here. The “options” finally can for example be used to specify if the data of the operands should be interpreted as signed or unsigned data or if the result shall be rounded or saturated.

Simulator

Since the ePUMA has not been developed as a chip yet the programming of the ePUMA has been done with help of a simulator, which had been developed at the ISY department. The simulator is python based and allows for cycle-accurate testing of the kernels for the sleipnir co-processors and the senior master processor. Furthermore the simulator takes the pipeline structure of the processors into
account so that it can help with debugging problems caused by data-dependencies. If for example an intermediate data result is used too early, before a previous instruction has been completely finished, it will show up as a wrong result in the simulator. This is very convenient for the programmer since there are many different instructions which require different numbers of pipeline stages and thus data dependency problems can easily occur. This is especially the case when aiming for an implementation with a fast kernel-execution time and including the lowest number of nop operations as possible.

**Notation**

Due to the fact that the sleipnir assembly instructions can include multiple options and up to three addresses with potentially complicated addressing patterns, the sleipnir code-lines tend to get relatively long. Therefore we will introduce a shorter notation for the code examples presented in this thesis as exemplified in listing 5.1. The first thing we should note here is the indentation in line 2. Whenever a line is indented in any sleipnir code example presented in this report, it means, that this line is a continuation of the instruction of the previous line in the code.

**Listing 4.4. Sleipnir code-example notations**

```
1 cmulwww <rnd, ss, mul=2, conj> m1 [ ar1+=s ]. v
2        v r f [ rar0 +=8 ]. v  m0 [ ar2 +=8 ]. v
3 cmulwww <..> m1 [ ... ]. v  v r f [ ... ]. v  m0 [ ... ]. v
4```

In order to avoid the need to divide the majority of the sleipnir instructions over multiple lines in our code examples, we introduce the place-holder “..” which represents the options or addressing patterns of the previous complete sleipnir instruction of the code example. The two instructions in Line 1 and 2 thus contain exactly the same sleipnir instruction as Line 4.

Now after we have looked into how the ePUMA platform and more specifically the sleipnir processors can be programmed, we will discuss in the following chapters the actual ePUMA implementations that have been developed in this thesis.
Chapter 5

FFT implementation

The following chapter will describe in detail how the 300 point IFFT has been implemented for the ePUMA. Matlab has been used as assisting tool for verifying all the computational steps, that needed to be executed. In the end of the chapter the achieved cycle cost of the FFT is evaluated and suggestions for possible improvements of the ePUMA architecture are made.

5.1 DFT and IDFT relation

We explained earlier that demodulating an LTE signal with 5MHz bandwidth requires both a 512 FFT and 300 point IFFT computation at the receiver side. While kernels for FFT calculations of sizes 16, 32, 64, 128, 512, 1024 and 2048 had already been implemented for the ePUMA, the 300 point IFFT needed to be derived from scratch. In the project work this task has been achieved by implementing a 300 point FFT, rather than an IFFT, because it is relatively simple to use an FFT algorithm for IFFT calculations, if the data is manipulated in a suitable way, as we will demonstrate in the following.

Let us first recall the equations for DFT and IDFT calculations. An N-point DFT of the in general complex input values $x_n$ can be calculated by:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-i2\pi k \frac{n}{N}}$$  \hspace{1cm} (5.1)

where $X_k$ is defined for $0 \leq k < N$ and represents the DFT of $x_n$. Accordingly an inverse discrete Fourier transformation is defined by the equation

$$x_k = \frac{1}{N} \sum_{n=0}^{N-1} X_n e^{i2\pi k \frac{n}{N}}$$  \hspace{1cm} (5.2)

where $x_k$ represents the IDFT of $X_n$.

An N-point IDFT can now be calculated by using an N-point DFT by executing the following steps:
1. Swap the imaginary and real parts of the complex input data.

2. Compute the N-point DFT

3. Swap the imaginary and real parts of the results

4. Divide the results by N

In the following we are going to proof that the above mentioned steps are equivalent to an IDFT calculation. Let us first define the mathematical operation “swap” of any complex data value \( Z = a + bi \) as follows,

\[
\text{swap}(Z) = \text{swap}(a + bi) = b + ai
\]  

(5.3)

We then want to proof what we described earlier in words that

\[
\frac{1}{N} \text{swap}[\text{DFT}(\text{swap}(x_n))] = \text{IFFT}(x_n)
\]  

(5.4)

where \( x_n \) represents any complex input data sequence of length \( N \).

Using equations 5.1 and 5.2 this means we want to show that:

\[
\frac{1}{N} \text{swap}\left(\sum_{n=0}^{N-1} \text{swap}(x_n) e^{-i2\pi k \frac{n}{N}}\right) = \frac{1}{N} \sum_{n=0}^{N-1} x_n e^{i2\pi k \frac{n}{N}}
\]  

(5.5)

The input data sequence \( x_n \) shall be represented by \( R(x_n) + i I(x_n) \) and obviously \( \text{swap}(x_n) \) is defined as \( I(x_n) + R(x_n) * i \). In order to simplify the notation, we define \( \alpha(n,k) = 2\pi k \frac{n}{N} \) and we furthermore need to recall Euler’s formula according to which \( e^{ai} = \cos(\alpha) + i \sin(\alpha) \). We can now simplify 5.5 to the following equation.

\[
\text{swap}\left(\sum_{n=0}^{N-1} \text{swap}(x_n)e^{-i\alpha}\right) = \sum_{n=0}^{N-1} x_n e^{i\alpha}
\]  

(5.6)

We will show that 5.6 is true by manipulating both sides seperately starting with the left side:

\[
\text{swap}\left(\sum_{n=0}^{N-1} (I(x_n) + R(x_n) * i)e^{-i\alpha}\right)
\]  

(5.7)

\[
= \text{swap}\left(\sum_{n=0}^{N-1} (I(x_n) + R(x_n) * i)(\cos(-\alpha) + \sin(-\alpha)i)\right)
\]  

(5.8)

\[
= \text{swap}\left[\sum_{n=0}^{N-1} (I(x_n)\cos(\alpha) + R(x_n)\sin(\alpha)) + (R(x_n)\cos(\alpha) - I(x_n)\sin(\alpha))i\right]
\]  

(5.9)

\[
= \sum_{n=0}^{N-1} (R(x_n)\cos(\alpha) - I(x_n)\sin(\alpha)) + (R(x_n)\sin(\alpha) + I(x_n)\cos(\alpha))i
\]  

(5.10)

Manipulating the right side of equation 5.6 gives us:
\[ \sum_{n=0}^{N-1} x_n e^{i\alpha} = \sum_{n=0}^{N-1} (R(x_n) + I(x_n) \ast i) e^{i\alpha} \] 
(5.11)

\[ = \sum_{n=0}^{N-1} (R(x_n) + I(x_n) \ast i)(\cos(\alpha) + \sin(\alpha)i) \] 
(5.12)

\[ = \sum_{n=0}^{N-1} (R(x_n)\cos(\alpha) - I(x_n)\sin(\alpha)) + (R(x_n)\sin(\alpha) + I(x_n)\cos(\alpha))i \] 
(5.13)

We can see that the result is identical with the expression in 5.9 and therefore we have shown, that it is possible to compute an IFFT by using an FFT as described above.

### 5.2 Fast Fourier Transformation

The fast fourier transformation (FFT) is, as the name implies, a quick method to compute a DFT. A more specific and generally accepted definition states, that all fast algorithms which can be used to calculate a DFT, are collectively known as FFT algorithms [11]. In this thesis a very common algorithm for FFT computation has been used, which is described in the following section.

#### 5.2.1 Cooley-Tukey FFT algorithm

In 1965 J. W. Cooley and J. W. Tukey revolutionized the DFT calculation by introducing a fast algorithm, which dramatically reduced the amount of computational steps to be performed for calculating DFTs, if their lengths were matching certain conditions [22]. Cooley and Tukey’s discovery led to several of the most widely studied FFT algorithms and is considered to be one of the greatest contributions to numerical analysis of the twentieth century [11, 22].

Let us consider any discrete input data sequence \( x_n \) with a length \( N \) on which an FFT shall be performed. We furthermore specify that the sequence length shall be non prime so that we can express \( N \) as a product of two integer numbers as \( N = a \ast b \) with \( a, b \neq 1 \). According to the Cooley-Tukey algorithm it is in this case possible to calculate the \( N \)-point DFT by

1. calculating a number of “\( a \)” DFTs of length “\( b \)” each,
2. \( N \) multiplications with so called twiddle-factors,
3. calculating a number of “\( b \)” DFTs of length “\( a \)”.

Before we describe this algorithm in more detail, we will give an example in the following for demonstrating why performing these three steps can significantly reduce the total number of computations compared to conventional DFT calculations.
Example 5.1: Algorithm cost evaluation for 12 point FFT

Let us consider an input data sequence of length \( N=12 \) for which we want to calculate the DFT. If we compute the DFT by using the convenient method, the sum in eq. (5.1) will have to be calculated 12 times. The total number of necessary complex additions and complex multiplications can thus be calculated as:

\[
\text{Operations} = 12 \cdot (11\text{add} + 12\text{mul}) = 132\text{add} + 144\text{mul} \quad (5.14)
\]

If we now evaluate the Algorithm cost for the Cooley-Tukey method for \( N=12=a\times b=3\times4 \), we will have to perform 3 DFT’s of length 4, 4 DFT’s of length 3 and 12 multiplications and the number of operations decreases to:

\[
\text{Operations} = 3 \times 4 \cdot (3\text{add}+4\text{mul}) + 4 \times 3 \cdot (2\text{add}+3\text{mul}) + 12\text{mul} = 60\text{add} + 96\text{mul} \\
(5.15)
\]

As we can see, there is already a significant reduction in computational complexity for this relatively short FFT length of \( N=12 \). Evidently the decrease of necessary arithmetic operations will be even more dramatic when using the algorithm for larger DFT lengths where \( N \) can be expressed as a product of significantly smaller prime numbers.

Furthermore, can the principal of the algorithm be used more than one time if \( N \) can be factorized into more than two numbers. A 16-point FFT for example could first be split into 8-point and 2-point FFT computations and then the 8-point FFTs could further be split up into 4-point and 2-point FFTs and so on. We should mention here though that in practise a 4-point FFT is not split into two 2-point FFTs because no additional speed can be gained by this as will be explained in more detail later. A 16 point FFT would therefore typically be calculated by 2 layers of four 4-point FFTs means \( N=16 \) would be factorized into the non primenumbers \( a,b=4 \).

The table 5.1 below lists some examples of computational cost reductions for different FFT lengths when using the Cooley-Tukey algorithm. The number of multiplicators which \( N \) contains describes the number of layers of “shorter” FFT-processing phases, that are required to calculate the complete FFT.

In the following section we will explain the mathematical background for the Cooley Tukey algorithm and demonstrate how the algorithm can be implemented in a practical way. In order to simplify the notation we define a weighting kernel

\[
W_N = e^{-2\pi i / N}, \quad (5.16)
\]

which is constant for a specific FFT length \( N \) and which will help us to express the twiddle-factors in a convenient way. The equation (5.1) for DFT computation can then be expressed as:

\[
X_k = \sum_{n=0}^{N-1} x_n (W_N)^{kn}, \quad k \in [0, N - 1] \\
(5.17)
\]
Table 5.1. Computational cost reductions of Cooley Tukey algorithm

<table>
<thead>
<tr>
<th>N</th>
<th>16</th>
<th>64</th>
<th>256</th>
<th>300</th>
<th>1024</th>
<th>4096</th>
</tr>
</thead>
<tbody>
<tr>
<td>Factors</td>
<td>$4^4$</td>
<td>$4^3$</td>
<td>$4^4$</td>
<td>$5^2 \times 4^3$</td>
<td>$4^5$</td>
<td>$4^6$</td>
</tr>
<tr>
<td>Nr of layers</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>Nr of additions</td>
<td>240</td>
<td>4032</td>
<td>65280</td>
<td>89700</td>
<td>1047552</td>
<td>16773120</td>
</tr>
<tr>
<td>Nr of multiplications</td>
<td>256</td>
<td>4096</td>
<td>65536</td>
<td>90000</td>
<td>1048576</td>
<td>16777216</td>
</tr>
<tr>
<td>Nr of additions cooley</td>
<td>96</td>
<td>576</td>
<td>3072</td>
<td>3900</td>
<td>15360</td>
<td>73728</td>
</tr>
<tr>
<td>Nr of multiplications cooley</td>
<td>144</td>
<td>896</td>
<td>4864</td>
<td>6000</td>
<td>24576</td>
<td>118784</td>
</tr>
<tr>
<td>Reduction of additions</td>
<td>60%</td>
<td>85.7%</td>
<td>95.3%</td>
<td>95.7%</td>
<td>98.5%</td>
<td>99.6%</td>
</tr>
<tr>
<td>Reduction of multiplications</td>
<td>43.8%</td>
<td>78.1%</td>
<td>92.6%</td>
<td>93.3%</td>
<td>97.7%</td>
<td>99.3%</td>
</tr>
</tbody>
</table>

In accordance with the prerequisites for the algorithm $N = a \times b$ with $a, b \neq 1$. We can then introduce index replacements for $n$ and $k$ as shown below [22]:

$$n = a \times n_1 + n_2, \quad n_1 \in [0, b - 1], n_2 \in [0, a - 1]$$

$$k = b \times k_2 + k_1, \quad k_1 \in [0, b - 1], k_2 \in [0, a - 1]$$

Inserting the index replacements into (5.17) we get:

$$X_{bk_2+k_1} = \sum_{n_2=0}^{a-1} \sum_{n_1=0}^{b-1} x_{an_1+n_2} \cdot (W_N)^{(bk_2+k_1)(an_1+n_2)}$$

which we can further simplify to:

$$= \sum_{n_2=0}^{a-1} \sum_{n_1=0}^{b-1} x_{an_1+n_2} \cdot e^{-\frac{2\pi i}{ab}[bn_2k_2+abn_1k_2+an_1k_1+n_2k_1]}$$

$$= \sum_{n_2=0}^{a-1} e^{-\frac{2\pi i}{ab}n_2k_1} \sum_{n_1=0}^{b-1} x_{an_1+n_2} \cdot e^{-\frac{2\pi i}{ab}n_1k_1} \cdot 1 \cdot e^{-\frac{2\pi i}{ab}n_2k_2}$$

$$= \sum_{n_2=0}^{a-1} (W_n)^{n_2k_1} \left[ \sum_{n_1=0}^{b-1} x_{an_1+n_2} \cdot (W_b)^{n_1k_1} \right] \cdot (W_a)^{n_2k_2}$$

(5.21)

With (5.21) we developed a mathematical expression that leads to the Cooley-Tukey algorithm [22]. We can easily verify by comparing with (5.17), that calculating the inner sum in (5.21) for $b$ times is equivalent to computing one FFT of length $b$ and that determining the outer sum for a number of $a$ times corresponds to one FFT calculation of length $a$. As mentioned earlier, the Cooley-Tukey algorithm reduces the computational complexity for one N-point FFT to the complexity of “$a$” $b$-point FFT calculations, N twiddle factor multiplications and “$b$” $a$-point FFT computations. This can be verified from (5.21) as follows:
1. We note that the complete N-point FFT calculation requires N computations of (5.21). Since the equation includes exactly one twiddle-factor multiplication, we can easily confirm that N multiplications are required for the complete FFT. In order to be more precise we should also note that whenever \( n_1 \) or \( k_1 \) is equal to zero \((W_n)^{n_1k_1}\) becomes equal to one and therefore the number of necessary multiplications with twiddle-factors decreases to \((a - 1) \ast (b - 1)\).

2. Similarly we observe that (5.21) contains exactly one outer-sum computation, which confirms that a number of "b" a-point FFTs is required for the complete FFT calculation in total.

3. The inner sum on the other hand gets calculated more than once namely "a" times inside (5.21) which is equivalent to a number of \(a^2 \ast b\)-point FFT computations in total. We can however observe that these b-point FFTs calculation will result in completely identical results for same values of \(k\). With \(k \in [0, a - 1]\) every a’th FFT will be identical and therefore the total number of required b-point FFT calculations reduces to \(a\).

In order to derive a step by step Cooley-Tukey algorithm from (5.21) it is convenient to interpret the in and output data as two-dimensional matrices of dimensions \(a \ast b\). The a- and b-point FFTs can then be executed on the rows and columns of the processing matrix respectively and the twiddle-factors multiplication can be illustrated as an elementwise multiplication of the processed matrix with a matrix \(T\) containing the twiddle factors. \(T\) as well is of dimensions \(a \ast b\) and the twiddle factors in the matrix can be expressed as below. As we can see every element in \(T\) can be calculated according to \(t_{a,b} = W^{(a-1)(b-1)}\).

\[
T_{a,b} = \begin{pmatrix}
t_{1,1} & t_{1,2} & \cdots & t_{1,b} \\
t_{2,1} & t_{2,2} & \cdots & t_{2,b} \\
\vdots & \vdots & \ddots & \vdots \\
t_{a,1} & t_{a,2} & \cdots & t_{a,b}
\end{pmatrix} = \begin{pmatrix}
W^{0 \ast 0} & W^{0 \ast 1} & \cdots & W^{0 \ast (b-1)} \\
W^{1 \ast 0} & W^{1 \ast 1} & \cdots & W^{1 \ast (b-1)} \\
\vdots & \vdots & \ddots & \vdots \\
W^{(a-1) \ast 0} & W^{(a-1) \ast 1} & \cdots & W^{(a-1) \ast (b-1)}
\end{pmatrix}
\]

\[
= \begin{pmatrix}
1 & 1 & \cdots & 1 \\
1 & W^1 & \cdots & W^{(b-1)} \\
\vdots & \vdots & \ddots & \vdots \\
1 & W^{(a-1)} & \cdots & W^{(a-1) \ast (b-1)}
\end{pmatrix}
\]

In the following we will assume an example of a 12 point FFT on the input data \(X_n\) for illustrating how the Cooley-Tukey algorithm can be implemented in a practical way.

1. Arrange the 12 input data values columnwise \(X_n\) in a 3 by 4 matrix as illustrated below. We refer to this matrix as \(M\) in the following.
5.2 Fast Fourier Transformation

\[ X_n \rightarrow \begin{pmatrix} X_1 & X_5 & X_9 \\ X_2 & X_6 & X_{10} \\ X_3 & X_7 & X_{11} \\ X_4 & X_8 & X_{12} \end{pmatrix} = M \]

2. Compute a 3-point FFT for each row of \( M \). We define the resulting values to be named as \( \tilde{X}_n \) and the resulting Matrix as \( \tilde{M} \).

\[ M = \begin{pmatrix} X_1 & X_5 & X_9 \\ X_2 & X_6 & X_{10} \\ X_3 & X_7 & X_{11} \\ X_4 & X_8 & X_{12} \end{pmatrix} \xrightarrow{FFT_s} \begin{pmatrix} \tilde{X}_1 & \tilde{X}_5 & \tilde{X}_9 \\ \tilde{X}_2 & \tilde{X}_6 & \tilde{X}_{10} \\ \tilde{X}_3 & \tilde{X}_7 & \tilde{X}_{11} \\ \tilde{X}_4 & \tilde{X}_8 & \tilde{X}_{12} \end{pmatrix} = \tilde{M} \]

3. Elementwise multiplication of \( \tilde{M} \) with the matrix \( T \), containing the twiddle factors. We define the operator \( \oplus \) to represent elementwise multiplication of matrices.

\[ \tilde{M} \oplus T = \tilde{M} \oplus \begin{pmatrix} 1 & 1 & 1 \\ 1 & w & w^2 \\ 1 & w^2 & w^4 \\ 1 & w^3 & w^6 \end{pmatrix} = \begin{pmatrix} \tilde{X}_1 & \tilde{X}_5 & \tilde{X}_9 \\ \tilde{X}_2 & \tilde{X}_6w & \tilde{X}_{10}w^2 \\ \tilde{X}_3 & \tilde{X}_7w^2 & \tilde{X}_{11}w^4 \\ \tilde{X}_4 & \tilde{X}_8w^3 & \tilde{X}_{12}w^6 \end{pmatrix} \]

4. Compute a 4-point FFT for each column of the matrix \( \tilde{M} \oplus T \). The resulting matrix \( Y_{out} \) contains the scrambled output sequence \( Y_n = FFT(X_n) \).

\[ \begin{pmatrix} \tilde{X}_1 & \tilde{X}_5 & \tilde{X}_9 \\ \tilde{X}_2 & \tilde{X}_6w & \tilde{X}_{10}w^2 \\ \tilde{X}_3 & \tilde{X}_7w^2 & \tilde{X}_{11}w^4 \\ \tilde{X}_4 & \tilde{X}_8w^3 & \tilde{X}_{12}w^6 \end{pmatrix} \xrightarrow{FFT_s} \begin{pmatrix} Y_1 & Y_2 & Y_3 \\ Y_4 & Y_5 & Y_6 \\ Y_7 & Y_8 & Y_9 \\ Y_{10} & Y_{11} & Y_{12} \end{pmatrix} = Y_{out} \]
5. Read out the data contained in $Y_{out}$ on a row by row basis for obtaining the ordered output sequence $Y_n$.

\[
\begin{pmatrix}
Y_1 & Y_2 & Y_3 \\
Y_4 & Y_5 & Y_6 \\
Y_7 & Y_8 & Y_9 \\
Y_{10} & Y_{11} & Y_{12}
\end{pmatrix} \rightarrow Y_1, Y_2, Y_3, ..., Y_{12}
\]

While the implementation of the algorithm is as we can see relatively simple for FFTs consisting of two layers, it can get rather complicated for FFTs consisting of multiple layers.

We can conclude from the previous discussion, that the main challenges in implementing the 300 point FFT where suitable data arrangements in all layers, which allowed for parallel vector computations, on the one hand, and rapid implementations of 5-point and 3-point FFTs (radix-5 and radix-3 computation kernels) on the other hand.

### 5.2.2 Radix-4 and Radix-2 FFTs

In Table 5.1 we gave some example for computational costs savings achieved by using the Cooley Tukey algorithm. However we still need to discuss another aspect of the algorithm, which concerns FFTs of length two and four, and which leads to even bigger computational cost savings whenever certain layers consist of smaller FFTs of these two specific lengths. If we take another look at equations (5.16) and (5.17) for calculating an N-point FFT we can easily verify, that if $N = 2$ or $N = 4$ the exponential factor $(W_N)^{kn} = e^{-2\pi i kn/N}$ can in all cases be simplified to one of the following values: $e^0 = 1$, $e^{i\pi/2} = i$, $e^{i\pi} = -1$ and $e^{3i\pi/2} = -i$. This means that for FFTs of lengths two or four, the multiplications with the term $(W_N)^{kn}$ can be replaced by additions and subtractions of the real and imaginary parts of the input data, which obviously will reduce the computational complexity even further. Due to this property the most rapid Cooley-Tukey algorithms that can be implemented are the so called radix-4 and radix-2 FFTs for FFT lengths where N can be factorized exclusively into the numbers 4 or 2 or combinations of the two. For this reason the computation of for example a 300 point FFT ($N = 5^2 * 3 * 4$) cannot be executed at the same speed as a 256-point ($N = 4^4$) or even a 512-point FFT ($N = 4^4 * 2$). We should furthermore note here that the radix-4 and radix-2 computations in FFT computations are typicaly also referred to as butterfly operations due to the shape of the data-flow diagram with which the FFTs can be illustrated.

### 5.3 300 point FFT Implementation for Sleipnir

The 300 point FFT implementation for the sleipnir SIMD processor was a major task of this thesis work. When implementing the 300 point FFT for the sleipnir processor Matlab has been used as assisting tool throughout the process. It was used for development and verification of the Cooley Tukey algorithm, computation
of twiddle factors, and verifying the intermediate results during the algorithm implementation for the SIMD processor. The algorithm has been developed for the input data type "complex word", means for complex data with 16 bit real and imaginary parts, so that four complex data values can be stored in one 128-bit vector.

5.3.1 Cycle cost estimations

In this section we will calculate a lower bound for the clock cycle time needed for running the 300 point FFT-algorithm on the sleipnir. As basis for this calculation we will consider the pure processing time of the computational instructions which were used in our approach towards this algorithm implementation. Any additional processing time caused by for example addressing operations, data movements between different memory components or delays due to data-dependencies will be neglected. This lower bound cycle cost estimation will serve us as a basis for evaluating the actual cycle cost that has been achieved for the sleipnir processor.

The 512 point FFT, which is needed as well in this thesis work, has got an execution duration of around 900 clock cycles. For the 300 point FFT, despite of its shorter length, a longer execution time was to be expected because of its three layers that do not consist of radix-4 or radix-2 instructions. Due to the butterfly instructions included in the sleipnir instruction set, a complete 4-point FFT (radix-4 operation) can be computed in merely one clock cycle effectively. We recall that the 300-point FFT algorithm consists of two layers of radix-5, one layer of radix-3 and one layer of radix-4 computations. The layers with radix-5 calculations were expected to add longer execution times to the algorithm than the other layers. Thus it has been decided to start with radix-5 computations in the first layer, as this is the only layer that does not require any multiplication with twiddle factors. The estimated cycle costs of the different layers will be discussed in the following.

First layer of FFT algorithm

With a total FFT length of 300, a number of 60 5-point FFTs had to be calculated in each of the first two layers of the algorithm. From (5.17) we can see that one 5-point FFT requires the computation of 25 complex multiplications (mulC) and 20 complex additions (addC). Taking into account all the cases where the factor \((W_N)^{kn}\) is equal to one (which is the case for \(n = 0\) or \(k = 0\)) the number of non trivial complex multiplications decreases to 16 and we get the following expression for the total number of complex operations for one 5-point FFT computation, which we refer to as \(opRadix5\):

\[
opRadix5 = 4 \times (4 \times mulC + 4 \times addC) + 4 \times addC \tag{5.24}
\]

Since the sleipnir is a SIMD processor, the computations should be executed on data vectors for optimum performance and speed. As mentioned earlier each vector can contain up to four complex data values referred to as complex words as it was illustrated in Figure 4.4. The instructions which were used for the complex
additions were addw or addwq depending on the fact if the situation required fast results or high parallelization. One addw instruction can handle the addition of four complex words per clock cycle. For the complex multiplications tcmacw instructions have been used, which can compute four complex word multiplications and accumulate these four results together. Thus tcmacw instructions can execute four complex word multiplications and three complex word additions per clock cycle. We can then compute the total number of instructions needed for the first layer of the FFT processing as:

\[
operPerRadix5 = 4 \left( 4 \ast \text{mulC} + 3 \ast \text{addC} \right) + \frac{8 \ast \text{addC}}{1 \text{ tcmacw instr.}} + \frac{8 \ast \text{addC}}{1 \text{ addw instr.}}
\]

(5.25)

\[
nrInst_{l1} = 60 \ast \left( 4 \ast \text{tcmacw} + 8 \ast \frac{\text{addw}}{8} \right) = 240 \ast \text{tcmacw} + 60 \ast \text{addw}
\]

(5.26)

Since each of these two instruction types included in the processing can be issued once per clock cycle our estimated minimum execution time for the processing in the first stage of the algorithm is equal to \(240 + 60 = 300\) clock cycles.

**Second layer of FFT algorithm**

The second layer of processing is very similar to the first layer since it contains as well 60 computations of 5-point FFTs. The execution time for this part can therefore also be estimated as 300. However, there is an additional computational step, which has to be performed in the second stage and that is the multiplication with twiddle factors. We could initially assume here that all 300 complex values would need to be multiplied with twiddle factors, however as mentioned earlier some of the twiddle factors are trivial having the value one. Since we are in the second stage of the algorithm with the second layer of radix-5 computations we can interpret the computations at this step as the calculation of a number of twelve 25-point FFT computations and thus the number of complex multiplications with twiddle factors reduces to \(12 \ast 4 \ast 4 = 192\) (compare (5.21)), which can be calculated by \(\frac{192}{4} = 48\) complex word vector multiplications. Thus the minimum estimated processing time for layer 2 is equal to \(300 + 48 = 348\) clock cycles.

**Third layer of FFT algorithm**

Layer three of the algorithm consists of a number of hundred radix-3 computations. From (5.17) we can easily verify that one radix-3 computation requires 9 complex multiplications and 6 complex additions in total. Again some of the multiplications are trivial, which reduces the total number of mulC operations. However for this layer it has been decided to combine the twiddle factors with the exponential factors used in the radix-3 computations. This means that these two factors have been multiplied together in advance, so that no extra multiplication with twiddle factors is necessary in this layer. Therefore the complex multiplications are only trivial in cases where both the twiddle-factor as well as the radix-3 exponent factor \((W_N)^{kn}\) are equal to one. This is the case for 3 of the nine multiplications,
5.3 300 point FFT Implementation for Sleipnir

reducing the total number of operations for one 3-point FFT computation to 6 complex multiplications and 6 complex additions.

Again sleipnir tmac-instructions, capable of complex multiplication and result accumulation have been used for the implementation. But since in this layer only two results of complex word multiplications need to be accumulated (and not four as in the radix-5 layers) the instruction tcmacdbw has been used instead. One of these instructions is able to execute two times the addition of the results of two complex word multiplications. Thus the instruction will be able to handle 4 complex word multiplications and 2 complex additions. We can now calculate the total number of instructions needed for layer 3 as follows:

\[
\text{operPerRadix}_3 = 6 \ast \text{mulC} + 6 \ast \text{addC} = 3 \ast (2 \ast \text{mulC} + 1 \ast \text{addC}) + 3 \ast \frac{1}{2} \text{tcmacdbw} + 3 \ast \frac{3}{8} \text{addw} \\
\text{nrInst}_3 = 100 \ast \left(3 \ast \frac{\text{tcmacdbw}}{2} + 3 \ast \frac{\text{addw}}{8}\right) = 150 \ast \text{tcmacdbw} + 38 \ast \text{addw}
\]

(5.27) (5.28)

We see that a total of 225 instruction is needed for the computations included in layer 3. Nevertheless the estimated cycle cost is higher than that, due to the fact that it is not possible to issue one tcmacdbw instruction in each clock cycle. Since we included the complex twiddle factor multiplications the number of complex multipliers is too large for being saved in the constant memory. Thus both source operands are fetched from LVMs and the destination is located in the LVMs as well. As mentioned before in each cycle it is only possible to access the LVMs exactly two times, one time for reading data from them and one time for writing data into them. Therefore the tcmacdbw instructions can merely be issued every second clock cycle and the estimated number of clock cycles for layer 3 computations increases to: \(\text{nrCycles}_L3 = 2 \ast 150 + 75 = 375\)

**Fourth layer of FFT algorithm**

Since this layer only consists of radix-4 operations it was relatively straightforward to be implemented for the sleipnir. As mentioned earlier one complete radix-4 operation can be calculated by just one butterfly operation in the ePUMA which are referred to as cr4bf in the sleipnir assembly code (cr4bf stands for complex radix-4 butterfly). Since layer 4 requires a number of 75 radix-4 computations the sleipnir is able to compute this layer with 75 instructions. As it has been the case in layer 3, due to the limitations of the LVM accessibility, each of these instructions can only be issued every second clock cycle and thus the estimated cycle cost for layer 4 is equal to 150 clock-cycles.

**5.3.2 Assembly implementation**

The complete assembly code of the 300-point FFT algorithm, that has been developed for the sleipnir can be seen in appendix X. Since the code got relatively long it has been split up into four parts according to the different layers of the
FFT implementation

algorithm. Due to the fact that it would take too long to discuss every single step of the kernel in detail, we will rather focus on explaining some specific code examples and the challenges, that were faced during the implementation.

The following code example 5.1 shows the tcmacw instructions which have been used in layer 1. The repeat command in line 1 means, that the following 15 lines will be executed four times. Thus we can easily verify that a total number of 240 tcmacw instructions will be issued in this code example, which matches the estimation for layer 1 in the previous section. The operands, which are saved in the LVM referred to by m0, are the input data values to the FFT algorithm. The operands stored in the constant memory have been precalculated in Matlab and represent the exponential factors \((W_N)^{kn}\) necessary for radix-5 computations.

We can see that this section does not include a single nop operation and that one tcmacw instruction is issued every clock cycle. Therefore this part of the FFT algorithm, which represents a big part of the total computational cost, could not have been implemented in a faster way for the sleipnir processor. The possibility of issuing one instruction every clock cycle is achieved by using the constant memory for one of the operands rather than the LVM for both operands. Therefore the LVM limitation of being only accessible twice per clock cycle does not affect the number of instructions per clock cycle in a negative way.

Listing 5.1. radix-5 computations in layer1

```
repeat 15 4
5*tcmacw <clr, ss, sat, div=1>
  m1[ar1+=2].d m0[ar0 +cm[car0+=1%]].v cm[r5twid1]
5*tcmacw <..> m1[ar1+=2].d m0[ar0 +cm[..]].v cm[r5twid2]
5*tcmacw <..> m1[ar1+=2].d m0[ar0 +cm[..]].v cm[r5twid3]
4*tcmacw <..> m1[ar1+=2].d m0[ar0 +cm[..]].v cm[r5twid4]
tcmacw <..> m1[ar1+=s].d m0[ar0+=8 +cm[..]].v cm[r5twid4]
9 5*tcmacw <..> m1[ar1+=2].d m0[ar0 +cm[..]].v cm[r5twid1]
10 5*tcmacw <..> m1[ar1+=2].d m0[ar0 +cm[..]].v cm[r5twid2]
11 5*tcmacw <..> m1[ar1+=2].d m0[ar0 +cm[..]].v cm[r5twid3]
12 4*tcmacw <..> m1[ar1+=2].d m0[ar0 +cm[..]].v cm[r5twid4]
tcmacw <..> m1[ar1+=s].d m0[ar0+=8 +cm[..]].v cm[r5twid4]
15 5*tcmacw <..> m1[ar1+=2].d m0[ar0 +cm[..]].v cm[r5twid1]
16 5*tcmacw <..> m1[ar1+=2].d m0[ar0 +cm[..]].v cm[r5twid2]
17 5*tcmacw <..> m1[ar1+=2].d m0[ar0 +cm[..]].v cm[r5twid3]
18 4*tcmacw <..> m1[ar1+=2].d m0[ar0 +cm[..]].v cm[r5twid4]
tcmacw <..> m1[ar1+=s].d m0[ar0+=s% +cm[..]].v cm[r5twid4]
```

We should also note here that in order to allow for a faster execution time, the input data to the FFT algorithm had to be loaded into the LVM with four gaps of 32 bits which separated the input data values into five blocks of 60 complex data-values each. This was necessary for circumventing the need of rearranging
the data before algorithm execution in order to allow for vectorized instructions without memory-bank conflicts. Obviously the data had to be arranged in suitable ways as well in between different layers in order to satisfy both memory bank requirements of the output of the previously processed layer as well as of the input data to the following processing layer.

The following listing 5.1 contains the complete implementation of the fourth layer of the algorithm which contains the 75 radix-4 butterfly instructions. As we can see the actual cycle cost matches the estimated cycle cost exactly for this layer. When we compare this code of the fourth layer with the overall length of the 300 point FFT algorithm, it becomes obvious that the implementation of FFT algorithms containing non-radix-4/radix-2 layers is a significantly bigger challenge for the sleipnir than implementing radix-2/radix-4 FFT algorithms.

### Listing 5.2. layer4 implementation

```
repeat 2 75

cr4bf<ss, rnd, sat> m1[ar0+=2 + cm[shift14]].v
m0[ar1+=2 + cm[shift14]].v m1[ar2+=8].v

1*nop
```

### 5.3.3 Cycle costs

The lower bound for the cycle cost on the sleipnir can be calculated by summing up the previously estimated cycle costs of the four layers different layers as follows:

\[
\text{Lowerbound} = 300 + 348 + 375 + 150 = 1173 \quad (5.29)
\]

In the result chapter we will present the actual cycle cost that has been achieved for the FFT implementation and discuss the various factors that limit the execution speed of the processor.
Chapter 6

Implementing the channel equalization

In the following chapter we will describe how the channel estimation and zero-forcing equalization has been implemented for the sleipnir processor. The algorithm has been realized by a number of consecutive computational steps that will be presented in the following.

6.1 Implementing the computation of the channel estimate

In order to calculate the complex channel estimates it was first necessary to extract from the received signal the complex reference symbols transmitted in the fourth OFDM symbol. We shall denote the received reference symbol sequence of length 300 as $Y_{\text{ref}}^k$. The transmitted reference symbol sequence $X_{\text{ref}}^k$ is known at the receiver side. We can thus calculate the complex channel estimates $\hat{H}^k$ as:

$$\hat{H}^k = \frac{Y_{\text{ref}}^k}{X_{\text{ref}}^k} \quad k \in [0, 299] \quad (6.1)$$

Instead of dividing the received symbols by the known transmitted reference symbols, which would have required 300 complex divisions, the reciprocals of the Reference sequence $(X_{\text{ref}}^k)^{-1}$ have been pre-calculated in Matlab, and thus the computation of the 300 complex channel estimates can be reduced to 300 complex multiplications.

$$\hat{H}^k = Y_{\text{ref}}^k \cdot (X_{\text{ref}}^k)^{-1} \quad k \in [0, 299] \quad (6.2)$$

The reciprocals of the reference symbol sequence have been loaded into the local vector memory before starting the algorithm execution. Since the sleipnir’s vector operations support 4 complex multiplications per clock cycle we might assume here, that the computation of the 300 complex channel estimates could be realized in 75 clock cycles. The listing 6.1 shows the actual sleipnir kernel implementation.
Implementing the channel equalization

Listing 6.1. Channel estimate computation

```c
channel_estimation:

   copy ar0c cm[AR0b]
   dcopy ar1c cm[AR1b]
   dcopy ar2c cm[AR2b]

   4*nop

   repeat 2 37
   cmulwww <rnd, ss, mul=2>
      m0[ar0++]8].v m1[ar1++]8].v m0[ar2++]8].v
   nop

   cmulwww <..> m0[ar0++]4].h m1[ar1++]8%.h m0[ar2++]4].h
   nop

   repeat 2 37
   nop

   cmulwww <..> m0[ar0++]8].v m1[ar1++]8].v m0[ar2++]8].v
   nop

   cmulwww <..> m0[ar0++]4%.h m1[ar1++=s%].h m0[ar2++]4%.h
```

After this code has been executed, the LVM referred to by m0 contains the 300 complex channel estimates $\hat{H}_k$. We can observe from the code, that a total of 76 complex multiplication instructions are executed instead of 75. We needed one instruction more because two half word-instructions had to be included. The reason for this is, that there exists a gap of half a vector in the middle of the data after the 512-point FFT algorithm execution, which is needed in order to avoid memory bank conflicts. Additionally we can see that a nop instruction is required between two consecutive cmulwww instructions which consequently increases the cycle cost by a factor of two. As it already has been the case for the FFT implementation, the limiting factor here is, that only two LVMs can be accessed per clock cycle, one for reading and one for writing. And since the reference symbol sequence is with a length of 300 too large for being stored in the constant memory, both source operands as well as the destination of the cmulwww instructions have to be LVMs, which makes the nop operations necessary. In conclusion the channel estimate computation requires 152 clock cycles plus a few extra cycles for addressing purposes.

6.2 Implementing the zero-forcing equalizer

For the equalization of the received data-symbols, there are different alternatives how the channel estimates can be used. The simplest approach, which has been
used in this thesis, is to assume the channel estimates to remain relatively con-
stant over the duration of one time-slot. This means that the channel estimates,
calculated by using the reference symbols in the fourth OFDM symbol of the slot,
will be used for equalization of the six surrounding OFDM symbols of the slot,
which contain data. Alternative enhanced channel estimation methods would in-
clude averaging or interpolation between consecutive channel estimates in order to
potentially gain even more accurate channel estimates for the equalization of the
OFDM data symbols. Naturally the benefit of these different methods depends on
the channel conditions and the coherence time of the channel in particular. Both
averaging and interpolation methods have not been considered in this thesis. It is
worth mentioning, that the approach that has been chosen for the project imple-
mentation can be considered sufficient for the majority of radio-channel scenarios.
Since the time duration of a slot is with 0.5 ms relatively short, interpolation
between channel estimates of consecutive slots for instance can only be expected
to give substantial throughput gains in certain special scenarios like for exam-
ple for very fast moving mobile users, which would correspond to a very short
channel-coherence time.

The zero-forcing equalization can then be realized according to the following
equation,

\[ \hat{X}_k = Y_k \ast H_k^{-1} = Y_k \ast \frac{1}{|H_k|^2} \ast \bar{H}_k \]

(6.3)

where \( \hat{X}_k \) represents the equalized complex symbol sequence, and \( Y_k \) denotes
the received data symbol sequence that shall be equalized.

The term referred to as “equalization factor” in 6.3 only needs to be calculated
one time for each slot, and its result will then be multiplied with the data sequences
of all six OFDM-Data symbols within the slot. In the sleipnir the computation of
this equalization factor has then been realized by executing the following steps:

1. Compute the 300 complex channel estimates \( \hat{H}_k \)
2. Calculate the complex conjugate of the estimates \( \bar{H}_k \)
3. Calculate the square of the absolute value of the estimates \( |\hat{H}_k|^2 \) by multi-
    plying \( \hat{H}_k \) with \( \bar{H}_k \)
4. Computing the reciprocal of the result, \( \frac{1}{|H_k|^2} \)
5. Multiplying the complex conjugates \( \bar{H}_k \) with the reciprocals \( \frac{1}{|H_k|^2} \)

While the implementation of most of these steps is relatively straightforward
for the sleipnir processor, the division in step 4 needs an additional algorithm,
since the sleipnir, as mentioned before, does not support instructions for division,
due to its fixed point architecture.
6.2.1 Kernel for 1/x

The division 1/x has been implemented with a kernel that has been investigated for the ePUMA SIMD co-processors in [19]. The kernel uses a polynomial approximation as follows:

\[
\frac{1}{x} \approx 1.2307 - 1.5166y + 1.8691y^2 - 2.1248y^3 + 2.8013y^4 - 7.0989y^5 \quad (6.4)
\]

with \( y = x - 0.8125 \) and \( x \in [0.5, 1] \). For details regarding how this polynomial approximation can be derived the reader is referred to [19]. We can see that it is necessary to subtract the value 0.8125 from the input to the reciprocal algorithm, and that the approximation is valid for input values within the interval 0.5 to 1. In our case for problem simplification we chose a channel transfer function, which allowed the results from the 512-point FFT algorithm to be scalable in such a way that they lay inside this range between 0.5 and 1.

6.2.2 Assembly code and cycle cost

The following listing 6.2 shows how the computation of \( |\hat{H}_k|^2 \) has been implemented for the sleipnir processor, which corresponds to step 2 and 3 of the earlier listed algorithm steps.

```
Listing 6.2. Channel equalization 1

1      dcopy ar1c cm[AR0c]
2      dcopy ar3c cm[AR3a]
3
4      repeat 10 9 //execute the next 10 lines 9 times
5      8*dcopy vrf[r0=0,+8].v m0[ar0+=8].v
6      1*nop
7      cmulww <rnd,ss,mul=2,conj> m1[ar1+=s].v
8      vrf[r0+=8].v m0[ar2+=8].v
9      cmulww <..> m1[ar3+=s].v vrf[r0+=8].v m0[ar2+=8].v
10     cmulww <..> m1[ar1+=s].v vrf[r0+=8].v m0[ar2+=8].v
11     cmulww <..> m1[ar3+=s].v vrf[r0+=8].v m0[ar2+=8].v
12     cmulww <..> m1[ar1+=s].v vrf[r0+=8].v m0[ar2+=8].v
13     cmulww <..> m1[ar3+=s].v vrf[r0+=8].v m0[ar2+=8].v
14     cmulww <..> m1[ar1+=s].v vrf[r0+=8].v m0[ar2+=8].v
15     cmulww <..> m1[ar3+=s].v vrf[r0+=8].v m0[ar2+=8].v
16     ///////////end repeat (72 out of 75 vectors are finished)
17
18     3*dcopy vrf[r0=0,+8].v m0[ar0+=8].v
19     1*nop
20     cmulww <..> m1[ar1+=s].v vrf[r0=0,+8].v m0[ar2+=8].v
21     cmulww <..> m1[ar3+=s].v vrf[r0+=8].v m0[ar2+=8].v
```
The input to the algorithm are the 300 complex channel estimates, stored in the LVM “m0”. The algorithm first copies as many of these values as possible into the vector register file (VRF). Afterwards the values in the VRF are multiplied with the identical values that are still stored in the LVM “m0” with the option “conj” activated. This option means that the complex conjugate of the second operand will be used as input for the multiplication. We recall that for a complex Number $Z$ with its complex conjugate $\bar{Z}$ the following equation holds:

$$Z \cdot \bar{Z} = |Z|^2 \quad (6.5)$$

Thus after execution of the code in listing 6.2, the LVM “m1” contains the square of the absolute value of the channel estimates which is the input to the reciprocal computing algorithm that is presented in the next code example.

As we can see in listing 6.3 the data first gets scaled to match the input range between $[0.5, 1]$ before the value 0.8125 gets subtracted for polynomial approximation. The polynomial approximation is executed in the vector register file for eight complex values during each loop iteration. The algorithm uses the instruction mulfbwww which stands for “multiplication feedback”. This operation executes a multiplication in which one operand is the result of a multiplication in the previous cycle. This instruction is very suitable for polynomial computation, because no nop operations are necessary between computations of different exponents of the input value. After the mulfbwww instructions have been executed the VRF contains the values $y$, $y^2$, $y^3$, $y^4$ and $y^5$ for each of the eight input values $y = x - 0.8125$. Furthermore these values have been permuted in such a way that for each of the eight polynomials all its exponential values can be accessed simultaneously. The polynomial coefficients are stored in the constant memory (referred to as Coeff in the code) and the computation of the polynomials has been realized by usage of tmacw instructions.

Listing 6.3. Channel equalization 2

```
// scaling the data into the range [0.5, 1]
37*cmulwww <rnd, ss ,mul=2>
   m0[ar0+=8].v m1[ar1+=s+cm[v2]].v cm[ scale1]
4 cmulwww <..> m0[ar0+=8].v m1[ar2+=8+cm[v2]].v cm[ scale1]
7*nop
// subtract value 0.8125 for polynomial approximation
38*addw <s ,sat ,rnd ,div=0> m1[ar1+=8].v m0[ar2+=8].v cm[ subb]
// Setup addressing registers
 dcopy vr0 cm[ones]
dcopy ar0c cm[AR0d]
```
We should furthermore note here that this loop computes in total $38 \times 8 = 304$ values while only 300 equalization factors need to be calculated. The 4 additional values that get computed are garbage values that appear due to the vectorized operations of the algorithm, since in the last loop execution only 4 values are left to be computed. It might be possible to achieve a minimal improvement in computational cost if we would replace the last (the 38th) loop iteration with...
other code that only computes four values instead of eight. However the potential benefit in cycle cost would not even reach 10 clock cycles and thus it did not seem reasonable to introduce the redundant code lines that would have been necessary for this purpose.

6.3 Cycle cost

The total implementation cost of the channel equalization algorithm is 1854 cycles. This number consists of 159 cycles for computation of the channel estimates and 1695 cycles for the computation of the equalization factors.

We can see from this channel equalizer implementation that the restriction of having only two accessible LVM’s per clock cycle significantly limits the overall execution time of algorithms on the sleipnir-processor. As we will further discuss in the following chapter on the example of the 300 point FFT kernel, a larger constant memory or the possibility of accessing three LVMs per clock cycle, would greatly increase the overall execution speed of algorithms on the sleipnir processor.
Chapter 7

Results and discussion

In this chapter we will summarize the results of the implementations that have been made during this thesis work. We will compare the achieved cycle cost of the FFT kernel, with the lower bound that was derived in the earlier chapter. Furthermore we will discuss the main challenges of the algorithm utilizations for the ePUMA and propose different measures that could be taken to enhance the processors overall performance.

7.1 Cycle cost of FFT algorithm

We recall that by summing up the estimated cycle costs of the four layers included in the algorithm for computing the 300 point FFT we can determine the theoretical minimum cycle cost of the algorithm as

\[ \text{Lowerbound} = 300 + 348 + 375 + 150 = 1173 \]  

(7.1)

The cycle cost that has been achieved for the different layers in the sleipnir implementation is shown in table 7.1.

<table>
<thead>
<tr>
<th>Layer</th>
<th>Lower bound cycle cost</th>
<th>Achieved cycle cost</th>
<th>Overhead in percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>300</td>
<td>445</td>
<td>48.3</td>
</tr>
<tr>
<td>2</td>
<td>348</td>
<td>570</td>
<td>63.8</td>
</tr>
<tr>
<td>3</td>
<td>338</td>
<td>456</td>
<td>34.9</td>
</tr>
<tr>
<td>4</td>
<td>150</td>
<td>156</td>
<td>4.0</td>
</tr>
<tr>
<td>Sum</td>
<td>1136</td>
<td>1627</td>
<td>43.2</td>
</tr>
</tbody>
</table>

Table 7.1. Achieved cycle cost 300 point FFT

The additional execution time is primarily caused by extra instructions for addressing purposes and additional delays due to data-dependencies. Furthermore the algorithm required a number of other instructions that we did not include in the lower-bound estimation like for example bit-wise shifting operations or the
movements of data between different memory components like LVMs and VRF. We can see that the percentage in overhead varies significantly between the different processing layers. This does not come as a surprise since the amount of overhead compared to the lower bound increases with higher complexity of the processing layer regarding to computation, data arrangements and addressing patterns.

7.2 Evaluation of the results

In order to evaluate the performance of the FFT implementation, presented in this work, we are going to compare the achieved cycle cost to several other FFT implementations in the following. In [16] three non-power-of-two point FFTs have been implemented for the ePUMA with sizes 1152, 1200 and 1536. In order to compare the cycle cost in [16] with our FFT implementation we will first estimate which cycle cost we could expect if we would use the 300 point FFT implementation presented in this thesis, in order to compute a 1200 point FFT, as such an extension would be relatively straightforward.

Applying the Cooley-Tukey algorithm once more we would simply have to calculate a number of four 300 point FFTs and add one more layer of radix 4 operations. With the total FFT length of 1200, this means that we would need an extra number of 300 radix-4 (cr4bf) operations, which could be executed every second clock cycle. Assuming an additional overhead of 4% for this additional layer (equivalent to the overhead we saw for the radix-4 layer implemented in the 300 point FFT) we can compute an estimate for the total cycle cost of an 1200 FFT as follows:

\[
\text{Cyclecost(FFT1200)} \approx 4 \times (1627) + (300 \times 2) \times 1.04 = 6508 + 624 = 7132
\]  

(7.2)

The following table shows the cycle costs that have been achieved in [16] and also the performance for these FFTs on the T1 dedicated FFT co-processor, which has also been used in [16] for evaluating the FFT implementations.

<table>
<thead>
<tr>
<th>FFT size</th>
<th>1152</th>
<th>1200</th>
<th>1536</th>
</tr>
</thead>
<tbody>
<tr>
<td>ePUMA (one SIMD) cycle cost in [16]</td>
<td>4074</td>
<td>7657</td>
<td>4296</td>
</tr>
<tr>
<td>T1 dedicated FFT co-processor (FFTC)</td>
<td>3026</td>
<td>3063</td>
<td>3795</td>
</tr>
</tbody>
</table>

We can see that our estimated cycle cost of 7132 based on the 300 point FFT implementation made in this thesis is below the achieved cycle cost for a 1200 point FFT in [16], which gives an indication of the efficiency of the FFT implementation done in this thesis work. However, we have to keep in mind that the cycle cost is only an estimate and addressing cost may potentially be somewhat higher when implementing the complete 1200 point FFT in reality. As we can see the T1 dedicated FFT co-processor shows a better performance in cycle cost, but it also has got more than double the computing ability and resources than one of
7.2 Evaluation of the results

ePUMA’s SIMDs [16]. Due to this differences in computing power, the results still show a reasonable performance of the ePUMA as an FFT processing unit. For more details regarding the comparism between ePUMA and the FFTC processor, the reader is referred to [16].

From the cycle cost estimate for the 1200 FFT we can also develop a minimum clock frequency for the ePUMA processor as a baseband processing unit in LTE base stations.

In [20] the throughput requirements as illustrated in table 7.3 have been used in order to derive minimum requirement for FFT and IFFT processing units implementing the LTE standard.

<table>
<thead>
<tr>
<th>Standard</th>
<th>MIMO scheme</th>
<th>TP FFT</th>
<th>TP IFFT</th>
</tr>
</thead>
<tbody>
<tr>
<td>3GPP LTE-A uplink</td>
<td>4 x 4</td>
<td>427 MS/s</td>
<td>250 MS/s</td>
</tr>
<tr>
<td>Future extension for 3GPP with</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>large-scale MIMO</td>
<td>128 x 8</td>
<td>13.65 GS/s</td>
<td>500 MS/s</td>
</tr>
</tbody>
</table>

Table 7.3. Aggregate throughput (TP) performance requirements for 3GPP LTE-A uplink receiver [20]

We recall that an IFFT of size 1200 is part of the baseband processing chain of the biggest transmission bandwidth defined for LTE of 20 MHz. (compare table 3.1) As we can see in table 7.3, the required throughput of LTE uplink is 250 Mega symbols per second.

With the same approach as used in [20] we can calculate the maximum processing time $t_{max}$ for an FFT of length 1200 as:

$$t_{max} = \frac{1}{250 \times 10^6 \times 1/s} = 4.8 \mu s$$

(7.3)

Assuming that one FFT shall be processed by one of ePUMA’s SIMDs we can now calculate a rough estimate for the minimum clock period that the ePUMA should operate with in order to handle the IFFT processing for the biggest LTE bandwidth:

$$t_{clk} = \frac{4.8 \times 10^{-6}}{7132} \approx 6.73 \times 10^{-10} s$$

(7.4)

And the minimum required clock frequency for the ePUMA can then be calculated as:

$$f_{clk_{min}} = \frac{1}{6.73 \times 10^{-10}} \approx 1.486 GHz$$

(7.5)

With an equivalent computation, and using the clock cycle cost for our implemented FFT of length 300, we can determine the minimum clock frequency for processing a BW of 5 MHz to be $f_{clk_{min}} \approx 339 MHz$.

Using different SIMDs for processing various FFTs in parallel, it may however be possible to execute the necessary baseband computation steps with even lower clock frequencies for the ePUMA. Such an approach would naturally cause
extra delays and the question how far the clock frequency could be reduced via parallelization needs to be further investigated and is not covered by this thesis.

7.3 Challenges of the implementation

The major challenges of this FFT implementation for the ePUMA were related to the fact that the different processing layers consisted of different smaller FFT sizes, and thus the layers varied significantly in the way they had to be computed. This fact together with the limitations of the LVMs memory banks accessibility put a lot of different requirements on the data-arrangements throughout the whole algorithm. In order to fully utilize the vector-operation potential of the sleipnir a big amount of various data-permutation patterns had to be used which had to be stored in the CM. Furthermore the big number of different data-arrangements made many different re-initializations of the special purpose registers for addressing necessary. The big number of permutation patterns on the one hand and the various vectors for address-register initialization on the other hand caused the CM to fill up quite rapidly. In fact the 128 vectors-storage capacity of the CM turned out to be the bottle-neck during the algorithm-implementation.

Cyclic addressing was used in order to reduce the necessary amount of address-register re-initializations, however the fact that only one step size could be configured by the user limited the benefit of this method. The gain of the cyclic addressing methods with respect to address-register re-initialization efforts could be increased significantly if it would be possible to configure a bigger number of different step sizes for incrementing the address pointers. An option to achieve this would be to remove the standard step-sizes 1,2,4,8 in order to enable the programmer to specify five different step sizes during address-register initialization instead of only one. That would have been very beneficial during this algorithm implementation, especially since the step-sizes 1, 2 and 4 have almost never been used for LVM addressing, due to the fact that vector operations were used most of the time. The possibility of using five configurable step sizes would have reduced the amount of address-register re-initializations and freed up more CM capacity as well.

Additional available CM capacity brings another advantage namely that more processing data like for example in this case Twiddle factors could be stored in the CM. This can speed up the execution time because computational instructions including one CM operand can be executed twice as often as instructions requiring three LVM accesses for the two operands and the destination. Due to all previous mentioned facts it might be worth considering if a bigger CM memory might be more desirable for the sleipnir processor. Of course that would increase the cost of the processor as it would also be the case for another potential improvement which could be to make the LVMs accessible three times per clock-cycle.

In conclusion we can summarize, that the limitations of the LVMs accessibility per clock cycle and/or the relatively small size of the CM should probably be addressed if the ePUMA should work as a processing unit for computational tasks of high complexity like for example baseband processing algorithms. Even if a
larger CM and a better LVM accessibility should turn out to be too expensive improvements, it is strongly proposed to at least give the programmer the flexibility of specifying more different step sizes for the address register in order to relax the restraints on the sleipnir’s CM capacity.
Chapter 8

Conclusions

In this thesis a simplified model of the LTE uplink receiving chain has been implemented in Matlab and parts of it have been mapped onto the ePUMA platform. We however need to keep in mind that from our highly simplified model there is still a very long way to go to developing a complete working LTE uplink baseband receiver which could run in LTE base stations. The ePUMA indeed offers a strong computational power due to its multi-core SIMD approach and it seems a reasonable assumption, based on the implementations made in this thesis work, that the platform has the computational potential to handle many of the algorithms used in an LTE base station. However, since the design of an LTE uplink baseband receiver is a very challenging task, which includes a big number of different algorithms of high complexity, it seems questionable if a platform which only supports assembly language will offer a comfortable enough programming environment for software developers in order to finish such an enormous task in reasonable time. A higher level language would certainly be desirable when implementing a project of such a high complexity like an LTE baseband receiver. The problem however is that a high level language could not efficiently exploit the full potential of the ePUMA platform and this puts some restraints on the general potential of the ePUMA for projects of high complexity. Another problem is, that the strong competition in the telecommunication field leads to ongoing efforts by vendors to continuously improve the LTE networks performance. Since the LTE standard is constantly enhanced and improved and base stations are being upgraded all the time by new product releases in an iterative process of product improvements it might be a very challenging task for any vendor to completely change the hardware platform for the baseband processing modules in their base stations.

During the kernel implementation of the 300 point FFT for the SIMD processors a number of challenges had to be surpassed. A main problem was that the algorithm required a lot of different addressing patterns and thus the constant memory filled up rapidly. For this specific algorithm development the constant memory really appeared to be a bottle-neck of the sleipnir. Based on the experiences made during the implementation it became clear that the constant memory allocation used for initialization of special purpose address registers could
be significantly reduced if more than one step size would have been available for incrementing the pointers stored in the addressing registers. An easy achievable and very beneficial solution would be to allow the specification of five step sizes instead of having only one configurable step size and the four standard step sizes 1, 2, 4 and 8 since many of them have never or rarely been used in the whole algorithm implementation. Another issue worth considering might be to increase the constant memory size, because a bigger constant memory furthermore allows for faster algorithm computations in many circumstances. Since we need to keep in mind the fact that a large constant memory produces high cost, an alternative solution might be to include SIMDs with different CM sizes in order to provide a reasonable trade-off between reducing cost while simultaneously not limiting the platforms capabilities too much.

8.1 Future work

The work on the ePUMA processor at ISY is currently ongoing. With respect to LTE baseband processing algorithms a lot of different work items can be thought off. It would be possible to further improve the model of the LTE baseband receiver chain implemented in this thesis. Such improvements could for example include the development of an MMSE equalizer, or the support of a wider range of transmission bandwidths or investigating different methods of dividing the workloads between the SIMD processors. Furthermore it might be interesting to compare the ePUMAs computational performance and cost with the actual performance and costs of the hardware modules that are used by vendors in their base stations in order to evaluate the theoretical potential of the ePUMA. Independent of the fact, if the ePUMA one day will serve as a processing unit in LTE base stations, the platform surely has got a very interesting concept, and further investigation of its potential as LTE base station unit is encouraged, due to the platforms enormous computational potential on the one hand and due to the great variety of learning opportunities that such a project offers for students on the other hand.
Bibliography


Appendix A

Kernel for 300 point FFT

1 FFT300:
2 //Layer1
3 //
4
5 dcopy ar0c cm[AR0a]
6 dcopy ar1c cm[AR1a]
7 dcopy ar2c cm[AR1c]
8 dcopy ar3c cm[AR3a]
9 dcopy car0c cm[car0_init]
10 dcopy car1c cm[car1_init]
11
12 4*dcopy vrf[rar0=0, +=8].v m0[ar0+=8].v
13 4*dcopy vrf[rar0+=8].v m0[ar1+=8].v
14 3*addwq <s, sat, rnd, div=1>
15
16 addwq <..> vrf[rar0+=8].v vrf[rar1+=8].v m0[ar3+=8].v
17 3*addwq <s, sat, rnd, div=1> vrf[rar0+=8].v vrf[rar1+=8].v m0[ar3+=8].v
18 addwq <..> vrf[rar0+=8].v vrf[rar1+=8].v m0[ar3+=8].v
19
20 3*nop
21
22 2*addwq <..> ml[ar2+=s+cm[v1]].v vrf[rar0=0, +=8].v vrf[rar1=32, +=8].v
23 addwq <..> ml[ar2+=s%+cm[v1]].v vrf[rar0+=8].v vrf[rar1+=8].v
24 addwq <..> ml[ar2+=s+cm[v1]].v vrf[rar0].v vrf[rar1].v
25
26 3*dcopy vrf[rar0=0, +=8].v m0[ar0+=8].v
27 3*dcopy vrf[rar0+=8].v m0[ar1+=8].v
28
29 3*addwq <..>
30
31 3*nop
32
33 3*addwq <..>
34
35 2*addwq <..>
36
37 2*addwq <..>
38
39 3*addwq <..>
40
41 3*addwq <..>
42
43 3*addwq <..>
44
Kernel for 300 point FFT

37 cmulwww <..> vrf \[ r0+\text{=}8+cm[\text{car0}\text{=}1\%].v cm[r25twid4] m0[ar2\text{=}s].v
38 cmulwww <..> vrf \[ r0+\text{=}8].v cm[r25twid1] m0[ar1\text{=}s].v
39 cmulwww <..> vrf \[ r0+\text{=}8+cm[\text{car0}\text{=}1].v cm[r25twid2] m0[ar1\text{=}s].v
40 cmulwww <..> vrf \[ r0+\text{=}8+cm[\text{car0}\text{=}1].v cm[r25twid3] m0[ar1\text{=}s].v
41 cmulwww <..> vrf \[ r0+\text{=}8+cm[\text{car0}\text{=}1\%].v cm[r25twid4] m0[ar1\text{=}s].v
42 dcopy ar2c cm[car1\text{=}1] //AR2d
43 dcopy arlc cm[car1\text{=}1] //ARld
44 3*nop
tcmacw <clr, ss, sat> ml[ar3\text{=}s]. d cm[r5twid1] m0[ar0].v
45 tcmacw <..> ml[ar3\text{=}s]. d cm[r5twid1] vrf[ra0\text{=}0, +\text{=}8].v
46 3+*tcmacw <..> ml[ar3\text{=}s]. d cm[r5twid2] vrf[ra0\text{=}0, +\text{=}8].v
47 tcmacw <..> ml[ar3\text{=}s]. d cm[r5twid2] m0[ar0].v
48 tcmacw <..> ml[ar3\text{=}s]. d cm[r5twid2] vrf[ra0\text{=}0, +\text{=}8].v
49 3+*tcmacw <..> ml[ar3\text{=}s]. d cm[r5twid2] vrf[ra0\text{=}0, +\text{=}8].v
50 tcmacw <..> ml[ar3\text{=}s]. d cm[r5twid3] m0[ar0].v
51 tcmacw <..> ml[ar3\text{=}s]. d cm[r5twid3] vrf[ra0\text{=}0, +\text{=}8].v
52 tcmacw <..> ml[ar3\text{=}s]. d cm[r5twid3] vrf[ra0\text{=}0, +\text{=}8].v
53 tcmacw <..> ml[ar3\text{=}s]. d cm[r5twid3] vrf[ra0\text{=}0, +\text{=}8].v
54 tcmacw <..> ml[ar3\text{=}s]. d cm[r5twid4] vrf[ra0\text{=}0, +\text{=}8].v
55 tcmacw <..> ml[ar3\text{=}s]. d cm[r5twid4] vrf[ra0\text{=}0, +\text{=}8].v
56 tcmacw <..> ml[ar3\text{=}s]. d cm[r5twid4] vrf[ra0\text{=}0, +\text{=}8].v
57 3+*tcmacw <..> ml[ar3\text{=}s]. d cm[r5twid4] vrf[ra0\text{=}0, +\text{=}8].v
58 tcmacw <..> ml[ar3\text{=}s]. d cm[r5twid4] vrf[ra0\text{=}0, +\text{=}8].v
59 tcmacw <..> ml[ar3\text{=}s]. d cm[r5twid4] vrf[ra0\text{=}0, +\text{=}8].v
60 3+*tcmacw <..> ml[ar3\text{=}s]. d cm[r5twid4] vrf[ra0\text{=}0, +\text{=}8].v
61 tcmacw <..> ml[ar1\text{=}s]. d cm[r5twid3] m0[ar0].v
62 tcmacw <..> ml[ar1\text{=}s]. d cm[r5twid3] vrf[ra0\text{=}32, +\text{=}8].v
63 tcmacw <..> ml[ar1\text{=}s]. d cm[r5twid3] vrf[ra0\text{=}32, +\text{=}8].v
64 tcmacw <..> ml[ar1\text{=}s]. d cm[r5twid3] vrf[ra0\text{=}32, +\text{=}8].v
65 tcmacw <..> ml[ar1\text{=}s]. d cm[r5twid4] m0[ar0].v
66 tcmacw <..> ml[ar1\text{=}s]. d cm[r5twid4] vrf[ra0\text{=}32, +\text{=}8].v
67 tcmacw <..> ml[ar1\text{=}s]. d cm[r5twid4] vrf[ra0\text{=}32, +\text{=}8].v
68 3+*tcmacw <..> ml[ar1\text{=}s]. d cm[r5twid4] vrf[ra0\text{=}32, +\text{=}8].v
69 3+*nop
70 addwq <s, sat, rnd, div=1> vrf[cm[vrf1]].v
71 vrf[cm[vrf1]].v vrf[cm[vrf2]].v
72 addwq <..> vrf[cm[vrf3]].v vrf[cm[vrf3]].v vrf[cm[rese1]].v
73 addwq <..> vrf[cm[vrf5]].v vrf[cm[vrf5]].v vrf[cm[vrf6]].v
74 2*nop
dcopy arlc cm[car1\text{=}1] //AR1b
75 addwq <..> vrf[cm[vrf7]].v vrf[cm[vrf7]].v vrf[cm[vrf8]].v
76 addwq <s, sat, rnd> ml[cm[car1\text{=}1]].v vrf[cm[vrf1]].v vrf[cm[vrf3]].v
77 2*nop //opt
78 addwq <..> ml[cm[car1\text{=}1]].v vrf[cm[vrf5]].v vrf[cm[vrf7]].v
79 4*nop
80 // end repeat
81 //%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%/%%//%%%%%.nextSibling
82 dcopy ar0c cm[car0\text{=}1] //Ar0f
83 dcopy ar0c cm[car0\_initc]
dcopy arlc cm[AR1b]
84
85 3+asrwrq vrf[ra0\text{=}0, +\text{=}8].v m0[cm[car1\text{=}1]].v cm[vbb]
86 5+asrwrq vrf[ra0\text{=}1].v m0[ar0\text{=}s+cm[shift1]].v cm[vbb]
87 7+asrwrq ml[ar1\text{=}+8].v m0[ar0\text{=}s+cm[shift1]].v cm[vbb]
88 dcopy arlc cm[AR1f]
dcopy ar2c cm[AR1f]
dcopy ar0c cm[AR0ea]
dcopy ar3c cm[AR0ea]
89 1*nop
90 3+addw <s, sat, rnd> m0[cm[car1\text{=}1]].v vrf[ra0\text{=}8].v ml[cm[car0\text{=}1]].v
91 2+addw <..> m0[ar0\text{=}2+cm[car1]].v
92 ml[ar3\text{=}2+cm[car0]].v vrf[ra0\text{=}0, +\text{=}2].d
93 addw <..> ml[ar0\text{=}s+cm[car1]].v ml[ar3\text{=}s+cm[car0]].v vrf[ra0\text{=}2].d
94 repeat 2 3
89

103  \begin{lstlisting}
2*addw<br..>m0[ar0+=2+cm[car1]].v ml[ar3+=2+cm[car0]].v vrf[rar1+=2].d
addw<br..>m0[ar0+=s+cm[car1]].v ml[ar3+=s+cm[car0]].v vrf[rar1+=2].d

105

106

107 5*addw <..> m0[ar1+=s%+cm[shift]].v
ml[ar2+=s%+cm[shift]].v vrf[rar0=24].v
dcopy ar0c cm[AR0e]
dcopy ar3c cm[AR0e]

108 5*addw<> m0[ar1+=s%+cm[shift]].v ml[ar2+=s%+cm[shift]].v vrf[rar0=32].v
5*addw<> m0[ar0+=s%+cm[shift]].v ml[ar3+=s%+cm[shift]].v vrf[rar0=40].v
5*addw<> m0[ar0+=s%+cm[shift]].v ml[ar3+=s%+cm[shift]].v vrf[rar0=48].v

dcopy vr0 ml[608].v
dcopy vr1 ml[616].v
dcopy vr2 ml[624].v
dcopy vr3 ml[632].v
dcopy vr4 ml[640].v
dcopy vr5 ml[648].v
dcopy vr6 ml[656].v
dcopy ar1c cm[AR1g]
dcopy ar2c cm[AR1g]

109 5*addw<> m0[ar0+=s%+cm[shift]].v ml[ar3+=s%+cm[shift]].v vrf[rar0=56].v
5*addw<> m0[ar0+=s%+cm[shift]].v ml[ar3+=s%+cm[shift]].v vrf[rar0=64].v
5*addw<> m0[ar0+=s%+cm[shift]].v ml[ar3+=s%+cm[shift]].v vrf[rar0=72].v
5*addw<> m0[ar0+=s%+cm[shift]].v ml[ar3+=s%+cm[shift]].v vrf[rar0=80].v

110 5*addw<> m0[ar0+=s%+cm[shift]].v ml[ar3+=s%+cm[shift]].v vrf[rar0=88].v
dcopy ar0c cm[AR0f]
dcopy ar3c cm[AR0f]

111 5*addw<> m0[ar0+=s%+cm[shift]].v ml[ar3+=s%+cm[shift]].v vrf[rar0=96].v
5*addw<> m0[ar0+=s%+cm[shift]].v ml[ar3+=s%+cm[shift]].v vrf[rar0=104].v
5*addw<> m0[ar0+=s%+cm[shift]].v ml[ar3+=s%+cm[shift]].v vrf[rar0=112].v

112 \end{lstlisting}
Kernel for 300 point FFT

33 tcmacdbw <.> ml[ar0+=s+cm[w]].h ml[960].v m0[ar2+=s+cm[w2]].v
34 7*nop
35 tcmacdbw <.> ml[ar0+=2+cm[w]].h ml[968].v m0[ar1+=s+cm[w2]].v
36 tcmacdbw <.> ml[ar0+=s+cm[w]].h ml[976].v m0[ar2+=s+cm[w2]].v
37 tcmacdbw <.> ml[ar0+=2+cm[w]].h ml[984].v m0[ar1+=s+cm[w2]].v
38 tcmacdbw <.> ml[ar0+=2+cm[w]].h ml[992].v m0[ar2+=s+cm[w2]].v
39 tcmcw <.> ml[ar0+=2].d ml[1000].v m0[ar1+=s+cm[w2]].v
40 2*nop
42 dcopy ar1c cm[AR1h]
43 dcopy ar2c cm[AR2e]
44 3*nop
45 tcmacdbw <.> ml[ar0+=2+cm[w]].h ml[1008].v m0[ar2+=s+cm[w2]].v
46 tcmacdbw <.> ml[ar0+=s+cm[w]].h ml[1016].v m0[ar1+=s+cm[w2]].v
47 tcmacdbw <.> ml[ar0+=2+cm[w]].h ml[1024].v m0[ar2+=s+cm[w2]].v
48 tcmacdbw <.> ml[ar0+=s+cm[w]].h ml[1032].v m0[ar1+=s+cm[w2]].v
49 tcmacdbw <.> ml[ar0+=2+cm[w]].h ml[1040].v m0[ar2+=s+cm[w2]].v
50 tcmacdbw <.> ml[ar0+=s+cm[w]].h ml[1048].v m0[ar1+=s+cm[w2]].v
51 tcmacdbw <.> ml[ar0+=2+cm[w]].h ml[1056].v m0[ar2+=s+cm[w2]].v
52 7*nop
53 tcmacdbw <.> ml[ar0+=2+cm[w]].h ml[1064].v m0[ar1+=s+cm[w2]].v
54 tcmacdbw <.> ml[ar0+=s+cm[w]].h ml[1072].v m0[ar2+=s+cm[w2]].v
55 tcmacdbw <.> ml[ar0+=2+cm[w]].h ml[1080].v m0[ar1+=s+cm[w2]].v
56 tcmacdbw <.> ml[ar0+=s+cm[w]].h ml[1088].v m0[ar2+=s+cm[w2]].v
57 tcmacdbw <.> ml[ar0+=2+cm[w]].h ml[1096].v m0[ar1+=s+cm[w2]].v
58 tcmcw <.> ml[ar0+=2].d ml[1104].v m0[ar2+=s+cm[w2]].v
60 7*nop
61 addw <s, sat, rnd, div=0> m0[0].v ml[0].v vr0
62 addw <.> m0[8].v ml[8].v vr1
63 addw <.> m0[16].v ml[16].v vr2
64 addw <.> m0[24].v ml[24].v vr3
65 addw <.> m0[32].v ml[32].v vr4
66 addw <.> m0[40].v ml[40].v vr5
67 addw <.> m0[48].d ml[48].d vr6.0d
68 addw <.> m0[50].v ml[50].v vr0
69 addw <.> m0[58].v ml[58].v vr1
70 addw <.> m0[66].v ml[66].v vr2
71 addw <.> m0[74].v ml[74].v vr3
72 addw <.> m0[82].v ml[82].v vr4
73 addw <.> m0[90].v ml[90].v vr5
74 addw <.> m0[98].d ml[98].d vr6.0d
75 addw <.> m0[100].v ml[100].v vr0
76 addw <.> m0[108].v ml[108].v vr1
77 addw <.> m0[116].v ml[116].v vr2
78 addw <.> m0[124].v ml[124].v vr3
79 addw <.> m0[132].v ml[132].v vr4
80 addw <.> m0[140].v ml[140].v vr5
81 addw <.> m0[148].d ml[148].d vr6.0d
82 83 dcopy arlc cm[AR1h]
84 dcopy ar2c cm[AR2e]
85 dcopy ar0c cm[AR0g]
86 4*nop
87 7*asrwq vr[ar0+=8].v m0[ar3+=s+cm[shift]].v cm[vbb]
88 tcmacdbw <cl, ss, sat> ml[ar0+=2+cm[w2]].h ml[800].v m0[ar2+=s+cm[w22]].v
89 tcmacdbw <.> ml[ar0+=s+cm[w2]].h ml[808].v m0[ar1+=s+cm[w22]].v
90 tcmacdbw <.> ml[ar0+=2+cm[w2]].h ml[816].v m0[ar2+=s+cm[w22]].v
91 tcmacdbw <.> ml[ar0+=s+cm[w2]].h ml[824].v m0[ar1+=s+cm[w22]].v
92 tcmacdbw <.> ml[ar0+=2+cm[w2]].h ml[832].v m0[ar2+=s+cm[w22]].v
93 tcmacdbw <.> ml[ar0+=s+cm[w2]].h ml[840].v m0[ar1+=s+cm[w22]].v
94 tcmacdbw <.> ml[ar0+=2+cm[w2]].h ml[848].v m0[ar2+=s+cm[w22]].v
95 7*nop
96 tcmacdbw <.> ml[ar0+=s+cm[w2]].h ml[856].v m0[ar1+=s+cm[w22]].v
97 tcmacdbw <.> ml[ar0+=2+cm[w2]].h ml[864].v m0[ar2+=s+cm[w22]].v
98 tcmacdbw <.> ml[ar0+=s+cm[w2]].h ml[872].v m0[ar1+=s+cm[w22]].v
Kernel for 300 point FFT

```c
// Layer 4
6*nop
repeat 275
    crdbf<s, rnd, sat> m1[ar0+=2 + cm[shift14]].v
    m0[ar1+=2 + cm[shift14]].v m1[ar2+=8].v
1*nop
```