Efficient WiMAX Receiver Implementation on a Programmable Baseband Processor

Christian Axell
Mikael Brogsten

LiTH-ISY-EX--06/3858--SE
Linköping 2006
Efficient WiMAX Receiver Implementation on a Programmable Baseband Processor

Christian Axell
Mikael Brogsten

LiTH-ISY-EX--06/3858--SE
Linköping 2006

Supervisor: Anders Nilsson, Björn Sihlbom
Examiner: Dake Liu
Linköping 2006-11-01
Efficient WiMAX Receiver Implementation on a Programmable Baseband Processor

Christian Axell & Mikael Brogsten

WiMAX provides broadband wireless access and uses OFDM as the underlying modulation technique. In an OFDM based wireless communication system, the channel will distort the transmitted signal and the performance is seriously degraded by synchronization mismatches between the transmitter and receiver. Therefore such systems require extensive digital signal processing of the received signal for retrieval of the transmitted information.

In this master thesis, parts of an IEEE 802.16d (WiMAX) receiver have been implemented on a programmable baseband processor. The implemented parts constitute baseband algorithms which compensates for the effects from the channel and synchronization errors. The processor has a new innovative architecture with an instruction set optimized for baseband applications.

This report includes theory behind the baseband algorithms as well as a presentation of how they are implemented on the processor. An impartial evaluation of the processor performance with respect to the algorithms used in the reference model is also presented in the report.

Nyckelord
WiMAX, OFDM, Baseband, LeoCore.
Abstract

WiMAX provides broadband wireless access and uses OFDM as the underlying modulation technique. In an OFDM based wireless communication system, the channel will distort the transmitted signal and the performance is seriously degraded by synchronization mismatches between the transmitter and receiver. Therefore such systems require extensive digital signal processing of the received signal for retrieval of the transmitted information.

In this master thesis, parts of an IEEE 802.16d (WiMAX) receiver have been implemented on a programmable baseband processor. The implemented parts constitute baseband algorithms which compensates for the effects from the channel and synchronization errors. The processor has a new innovative architecture with an instruction set optimized for baseband applications.

This report includes theory behind the baseband algorithms as well as a presentation of how they are implemented on the processor. An impartial evaluation of the processor performance with respect to the algorithms used in the reference model is also presented in the report.
Acknowledgements
We would like to thank…

- Our industrial supervisor, Björn Sihlbom, for sharing his expertise in signal processing, and for support and encouragement.

- Our manager, Liselotte Wanhov, for taking good care of us.

- Our examiner, Dake Liu, for accepting us for this master thesis.

- Our university supervisor, Anders Nilsson, for guidance and encouragement.

- Eric Tell at Coresonic for helping us with processor related questions.

- Our opponents, Joakim Bjärmark and Marco Strandberg, for many useful comments on this thesis.
# Table of Contents

1 **Introduction** ............................................................................................................................. 1

1.1 Background .............................................................................................................................. 1
1.2 Goal and limitations................................................................................................................... 1
1.3 Disposition ............................................................................................................................... 1
1.4 Reading instructions .................................................................................................................. 1
1.5 Who should read this thesis? ................................................................................................... 1

2 **OFDM – Orthogonal Frequency Division Multiplexing** ............................................................ 2

2.1 OFDM transmission technique ................................................................................................. 3
2.2 Benefits....................................................................................................................................... 4
2.3 Disadvantages............................................................................................................................ 4
2.4 Usage ......................................................................................................................................... 4

3 **WiMAX** .................................................................................................................................... 5

3.1 WiMAX as a wireless solution for fixed broadband internet access ........................................ 5
3.2 IEEE 802.16d.............................................................................................................................. 6
3.2.1 Frequency domain ................................................................................................................... 6
3.2.2 Time domain ........................................................................................................................... 7
3.2.3 Preamble .................................................................................................................................. 7

4 **Reference model** ...................................................................................................................... 8

4.1 Transmitter ................................................................................................................................ 8
4.2 Channel ..................................................................................................................................... 9
4.3 Receiver ................................................................................................................................... 9
4.4 System parameters...................................................................................................................... 9
4.5 Quantization ............................................................................................................................... 9

5 **LeoCore technology** ................................................................................................................ 11

5.1 Single Instruction Multiple Task – SIMT .................................................................................. 11
5.2 Processor core .......................................................................................................................... 12
5.2.1 CMAC .................................................................................................................................... 12
5.2.2 ALU ....................................................................................................................................... 12
5.3 Memories .................................................................................................................................. 13
5.4 Instruction set ............................................................................................................................ 13
5.4.1 Vector instructions ................................................................................................................ 13
5.5 Coresonic Developer Studio ....................................................................................................... 15

6 **Angle calculations in hardware – CORDIC** ............................................................................. 16

6.1 Implementation ......................................................................................................................... 17
6.1.1 Performance .......................................................................................................................... 20
7 Baseband processing................................................................................................................. 21
7.1 Packet detection ..................................................................................................................... 21
7.2 The Discrete Fourier Transform – DFT .................................................................................. 23
  7.2.1 DFT implementation – FFT .......................................................... 23
  7.2.2 Radix-2 Sande-Tukey FFT algorithm ....................................................... 24
  7.2.3 Implementation ......................................................................................... 26
  7.2.3.1 Design flow ...................................................................................... 26
  7.2.4 Twiddle factors in CM ................................................................................. 27
  7.2.4.1 Performance .................................................................................... 27
7.3 Channel Estimation ................................................................................................................. 28
  7.3.1 Implementation ......................................................................................... 29
  7.3.1.1 LS-estimated frequency response samples ........................................ 29
  7.3.1.2 Phase ramp calculation ...................................................................... 30
  7.3.1.3 Symbol timing offset ......................................................................... 30
  7.3.1.4 Phase ramp compensation ................................................................. 31
  7.3.1.5 Channel estimate filtering ................................................................. 33
  7.3.1.6 Resource allocation ........................................................................... 36
  7.3.1.7 Performance .................................................................................... 37
7.4 Phase tracking ........................................................................................................................ 38
  7.4.1 Frequency offset ....................................................................................... 38
  7.4.2 Timing offset ........................................................................................... 38
  7.4.3 Time and frequency-domain effects .......................................................... 39
  7.4.4 Pilot subcarriers ...................................................................................... 40
  7.4.5 Implementation ....................................................................................... 41
  7.4.5.1 Pilot subcarrier phase rotation ......................................................... 41
  7.4.5.2 Frequency dependent phase ramp of pilot subcarriers .................. 42
  7.4.5.3 Time dependent phase ramp of pilot subcarriers .......................... 42
  7.4.5.4 Average phase rotation .................................................................... 43
  7.4.5.5 Phase difference of subcarriers with respect to time ...................... 43
  7.4.5.6 Phase difference between two subsequent subcarriers ................ 44
  7.4.5.7 Compensation of data subcarriers ..................................................... 45
  7.4.5.8 Resource allocation .......................................................................... 46
  7.4.5.9 Performance .................................................................................... 47
7.5 OFDM symbol demodulation............................................................................................... 48
  7.5.1 Demodulation example ........................................................................ 50
  7.5.2 Ramesh algorithm ................................................................................ 51
  7.5.3 Implementation ...................................................................................... 52
  7.5.3.1 Resource allocation ......................................................................... 53
  7.5.3.2 Performance ................................................................................... 54
8 Receiver integration ................................................................................................................. 56
8.1 Transmitter .......................................................................................................................... 57
8.2 Channel ................................................................................................................................. 57
8.3 Receiver ................................................................................................................................. 57
  8.3.1 Simulation results .................................................................................. 58
  8.3.2 Total resource allocation ...................................................................... 63
  8.3.3 Clock frequency .................................................................................... 64
9 Conclusions ............................................................................................................................... 65
9.1 Evaluation of LeoCore.......................................................................................................... 65
  9.1.1 Pros ....................................................................................................... 66
List of Figures

Figure 1 – Overlapping orthogonal subcarriers ................................................................. 2
Figure 2 – OFDM transmitter [2] ..................................................................................... 3
Figure 3 – OFDM receiver [2] .......................................................................................... 3
Figure 4 – WiMAX network topology .............................................................................. 5
Figure 5 – OFDM symbol frequency description ............................................................... 6
Figure 6 – OFDM symbol time structure ........................................................................... 7
Figure 7 – DL and network entry preamble structure ........................................................... 7
Figure 8 – Reference model ................................................................................................ 8
Figure 9 – Transmitter ....................................................................................................... 8
Figure 10 – Receiver ......................................................................................................... 9
Figure 11 – Principle view of LeoCore technology ............................................................. 11
Figure 12 – Network chain in LeoCore ............................................................................ 12
Figure 13 – Part of assembly program .............................................................................. 14
Figure 14 – Coresonic Developer Studio development platform ....................................... 15
Figure 15 – Vector rotation in the Cartesian coordinate system ....................................... 16
Figure 16 – CORDIC flowchart .................................................................................... 18
Figure 17 – Angle representation ..................................................................................... 20
Figure 18 – Baseband processing in the receiver ............................................................... 21
Figure 19 – Autocorrelation and energy based packet detection ....................................... 22
Figure 20 – 8-point decimation-in-frequency FFT signal flow graph ................................. 25
Figure 21 – Decimation-in-frequency butterfly operation .................................................. 25
Figure 22 – Channel estimation tasks .............................................................................. 28
Figure 23 – Channel frequency response samples for DC- and guard band subcarriers .......... 29
Figure 24 – Impulse response of FIR filter .................................................................... 33
Figure 25 – Magnitude response of FIR filter ................................................................. 34
Figure 26 – Direct form FIR filter structure .................................................................... 35
Figure 27 – Subcarrier phase rotation for adjacent OFDM symbols .................................. 39
Figure 28 – Phase tracking tasks ..................................................................................... 41
Figure 29 – BPSK constellation [6] .................................................................................. 49
Figure 30 – QPSK constellation [6] ................................................................................ 49
Figure 31 – 16-QAM constellation [6] ............................................................................ 49
Figure 32 – 64-QAM constellation [6] ............................................................................ 49
Figure 33 – Example of received symbol in a 16-QAM constellation ............................... 50
Figure 34 – Decision boundary I ≥ 0 ............................................................................. 50
Figure 35 – Decision boundary Q ≥ 0 ............................................................................. 50
Figure 36 – Decision boundary |I| < 2 ............................................................................. 51
Figure 37 – Decision boundary |Q| < 2 ............................................................................. 51
Figure 38 – Simulation .................................................................................................. 56
Figure 39 – Receiver architecture ................................................................................. 57
Figure 40 – QPSK constellation symbols before phase tracking and compensation ........... 58
Figure 41 – 16-QAM constellation symbols before phase tracking and compensation ........ 58
Figure 42 – QPSK constellation symbols after phase tracking and compensation in LeoCore ................................................................. 59
Figure 43 – QPSK constellation symbols after phase tracking and compensation in Matlab ................................................................................................. 59
Figure 44 – 16-QAM constellation symbols after phase tracking and compensation in LeoCore ......................................................................................... 59
Figure 45 – 16-QAM constellation symbols after phase tracking and compensation in Matlab ................................................................................................. 59
Figure 46 – Equalized QPSK constellation symbols in LeoCore ...................................... 60
Figure 47 – Equalized QPSK constellation symbols in Matlab ......................................... 60
Figure 48 – Equalized 16-QAM constellation symbols in LeoCore .................................. 60
Figure 49 – Equalized 16-QAM constellation symbols in Matlab ..................................... 60
Figure 50 – Histogram of soft bits from QPSK constellation symbols produced by LeoCore ................................................................................................. 61
Figure 51 – Histogram of soft bits from QPSK constellation symbols produced by Matlab ................................................................................................. 61
Figure 52 – Histogram of soft bits from 16-QAM constellation produced by LeoCore ........ 61
Figure 53 – Histogram of soft bits from 16-QAM constellation produced by Matlab ............ 61
List of Tables

Table 1 – Specified system parameters .......................................................................................................................... 9
Table 2 – SQR and corresponding number of correct bits ................................................................................................ 10
Table 3 – CORDIC table of arctangent values stored in IM ......................................................................................... 19
Table 4 – Twiddle factors in CM ...................................................................................................................................... 27
Table 5 – Complex exponential function look-up table .................................................................................................. 31
Table 6 – FIR filter taps .................................................................................................................................................. 34
Table 7 – Channel estimation performance ..................................................................................................................... 37
Table 8 – CM content: phase tracking and compensation ................................................................................................. 46
Table 9 – Performance of phase tracking and compensation .......................................................................................... 47
Table 10 – Performance of the demodulation implementation ........................................................................................... 54
Table 11 – Simulation results: QPSK signal ....................................................................................................................... 62
Table 12 – Simulation results: 16-QAM signal .................................................................................................................. 62
Table 13 – LeoCore memory usage .................................................................................................................................. 63
Table 14 – Total CM content .............................................................................................................................................. 63
Table 15 – Initial data memory content ............................................................................................................................. 66
Table 16 – Data memory content after moving data ....................................................................................................... 66
# Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC</td>
<td>Analog to Digital Converter</td>
</tr>
<tr>
<td>ADSL</td>
<td>Asynchronous DSL</td>
</tr>
<tr>
<td>ALU</td>
<td>Arithmetic Logic Unit</td>
</tr>
<tr>
<td>ASIC</td>
<td>Application Specific Integrated Circuit</td>
</tr>
<tr>
<td>BS</td>
<td>Base station</td>
</tr>
<tr>
<td>BPSK</td>
<td>Binay Phase-Shift Keying</td>
</tr>
<tr>
<td>BW</td>
<td>Bandwidth</td>
</tr>
<tr>
<td>CDS</td>
<td>Coresonic Developer Studio</td>
</tr>
<tr>
<td>CM</td>
<td>Coefficient Memory</td>
</tr>
<tr>
<td>CMAC</td>
<td>Complex MAC</td>
</tr>
<tr>
<td>CORDIC</td>
<td>COordinate Rotation Digital Computer</td>
</tr>
<tr>
<td>CP</td>
<td>Cyclic Prefix</td>
</tr>
<tr>
<td>DAC</td>
<td>Digital to Analog Converter</td>
</tr>
<tr>
<td>DFE</td>
<td>Digital Front End</td>
</tr>
<tr>
<td>DFT</td>
<td>Discrete Fourier Transform</td>
</tr>
<tr>
<td>DIF</td>
<td>Decimation-In-Frequency</td>
</tr>
<tr>
<td>DM</td>
<td>Data Memory</td>
</tr>
<tr>
<td>DSL</td>
<td>Digital Subscriber Line</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital Signal Processing</td>
</tr>
<tr>
<td>DTFT</td>
<td>Discrete-Time Fourier Transform</td>
</tr>
<tr>
<td>DVB-T</td>
<td>Digital Video Broadcasting - Terrestrial</td>
</tr>
<tr>
<td>FEC</td>
<td>Forward Error Correction</td>
</tr>
<tr>
<td>FFT</td>
<td>Fast Fourier Transform</td>
</tr>
<tr>
<td>FIR</td>
<td>Finite Impulse Response</td>
</tr>
<tr>
<td>IDFT</td>
<td>Inverse DFT</td>
</tr>
<tr>
<td>IEEE</td>
<td>Institute of Electrical and Electronics Engineers</td>
</tr>
<tr>
<td>ICI</td>
<td>Inter-Channel Interference</td>
</tr>
<tr>
<td>IM</td>
<td>Instruction Memory</td>
</tr>
<tr>
<td>ISI</td>
<td>Inter-Symbol Interference</td>
</tr>
<tr>
<td>I/Q</td>
<td>In-phase/Quadrature</td>
</tr>
<tr>
<td>LAN</td>
<td>Local Area Network</td>
</tr>
<tr>
<td>LS</td>
<td>Least Square</td>
</tr>
<tr>
<td>LSB</td>
<td>Least Significant Bit</td>
</tr>
<tr>
<td>MAC</td>
<td>Multiply And Accumulate</td>
</tr>
<tr>
<td>MAN</td>
<td>Metropolitan Area Network</td>
</tr>
<tr>
<td>MCM</td>
<td>Multi Carrier Modulation</td>
</tr>
<tr>
<td>MSB</td>
<td>Most Significant Bit</td>
</tr>
<tr>
<td>NLOS</td>
<td>Non Line Of Sight</td>
</tr>
<tr>
<td>OFDM</td>
<td>Orthogonal Frequency Division Multiplexing</td>
</tr>
<tr>
<td>OFDMA</td>
<td>Orthogonal Frequency Division Multiple Access</td>
</tr>
<tr>
<td>PM</td>
<td>Program Memory</td>
</tr>
<tr>
<td>QAM</td>
<td>Quadrature Amplitude Modulation</td>
</tr>
<tr>
<td>QPSK</td>
<td>Quadrature Phase-Shift Keying</td>
</tr>
<tr>
<td>Acronym</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>SDR</td>
<td>Software Defined Radio</td>
</tr>
<tr>
<td>SFO</td>
<td>Sampling clock Frequency Offset</td>
</tr>
<tr>
<td>SIMD</td>
<td>Single Instruction Multiple Data</td>
</tr>
<tr>
<td>SIMT</td>
<td>Single Instruction Multiple Task</td>
</tr>
<tr>
<td>SNR</td>
<td>Signal to Noise Ratio</td>
</tr>
<tr>
<td>SQR</td>
<td>Signal to Quantization Ratio</td>
</tr>
<tr>
<td>SS</td>
<td>Subscriber Station</td>
</tr>
<tr>
<td>T1</td>
<td>Digital Signal 1</td>
</tr>
<tr>
<td>VLIW</td>
<td>Very Long Instruction Word</td>
</tr>
<tr>
<td>WiMAX</td>
<td>Worldwide Interoperability for Microwave Access</td>
</tr>
<tr>
<td>WLAN</td>
<td>Wireless LAN</td>
</tr>
</tbody>
</table>
1 Introduction

1.1 Background
Wireless communication is a fast growing and changing market. New standards improve bandwidth and mobility and force the telecommunication companies to develop and/or change their entire systems. Wireless terminals need to handle several different standards like GSM, 3G, Bluetooth and WLAN. These demands lead to an increased interest in Software Defined Radio (SDR), i.e. radio devices reconfigurable with software during runtime. This leads to hardware reuse, lower cost for multiple radio standard support and extended life time via software updates. The evolution towards SDR opens the market for new companies with pioneering technologies and Coresonic AB develops programmable baseband processors for multimode modems. Coresonic provides silicon intellectual property to their customers and Ericsson AB has an interest in their processor architecture. Therefore Ericsson have initiated this master thesis for evaluation purposes.

1.2 Goal and limitations
The goal of this thesis is to implement a specified part of the baseband receiver algorithms on a LeoCore technology based processor. To keep the work load on a realistic level the implementation is limited to include the algorithms for packet detection, FFT, channel estimation, phase tracking, compensation, and demodulation. The work will not cover any modification or evaluation of the algorithms used in the reference model. No comparisons are made with any existing ASIC or DSP-processor solution for baseband processing.

1.3 Disposition
The work has mainly consisted of four phases. In the first phase we understood the technologies OFDM and WiMAX. In the second phase we understood the algorithms in the Matlab model. The third and most extensive phase was the implementation part, where we implemented the reference model algorithms as subroutines in the processor. In the fourth and last phase we did a simulation of the entire receiver where the different subroutines were integrated.

1.4 Reading instructions
- Chapter 2 describes the principles of OFDM.
- Chapter 3 presents WiMAX and describes the IEEE 802.16d standard.
- Chapter 4 presents the reference model.
- Chapter 5 describes the LeoCore technology.
- Chapter 6 presents how to calculate angles in the processor.
- Chapter 7 presents the theory behind the baseband algorithms and the implementation.
- Chapter 8 presents simulation results from the integration of the receiver.
- Chapter 9 presents our conclusions.

1.5 Who should read this thesis?
The intended reader has knowledge of DSP-processors, digital signal processing, and engineering knowledge equivalent to a fourth year Master of Science student. The thesis can be seen as an introduction to OFDM based standards in general and WiMAX as an application in particular.
2 OFDM – Orthogonal Frequency Division Multiplexing

OFDM is a transmission technique built for high speed bi-directional wired or wireless data communication. The technique is based upon the idea of multi-carrier modulation (MCM) where transmitted data is modulated on several orthogonal frequencies, called subcarriers, which are added together into a composite signal. Its history dates back to the 1960’s but it has not until recently become popular since economical high speed digital signal processing components has not been available.

In a single-frequency baseband channel, e.g. radio or television, data is transmitted on one main carrier frequency. In a multiple-frequency baseband channel, e.g. OFDM systems, data is transmitted concurrently on several orthogonal (sub) carrier frequencies. The subcarriers are closely spaced together but still orthogonal, which means that they are perpendicular in a mathematical sense, and do not interfere with each other. This can be seen in Figure 1, where the spectral peaks of each subcarrier coincide with zero crossing of all other subcarriers. This gives a symbol duration that is increasing proportionally to the number of subcarriers, which reduces the effects of intersymbol interference (ISI) in wireless communication caused by multipath interference, i.e. received reflections of the original signal. [1], [13].

![Overlapping orthogonal subcarriers](image_url)

Figure 1 – Overlapping orthogonal subcarriers
2.1 OFDM transmission technique

The principle of the construction of the OFDM signal is presented in Figure 2. Data, $s[n]$, is split into several parallel data paths that are mapped onto a symbol stream, $X_i$, where each stream represents a subcarrier with modulated data. The inverse FFT operation is used to create a complex discrete-time signal and the real and imaginary parts are separately converted to analog representation by the Digital to Analog Converters (DAC). The produced analog signals are quadrature-mixed and used to modulate the main carrier frequency wave, $f_c$, resulting in one in-phase, $I$, and one quadrature, $Q$, signal to enable transmission of the original complex signal. The I and Q signals are summed into a signal, $s(t)$, that is transmitted by the antenna.

![Figure 2 – OFDM transmitter [2]](image)

The principle of the receiver is presented in Figure 3. The received signal, $r(t)$, is quadrature-mixed down to baseband with the same carrier frequency as used in the transmitter. The real (I) and imaginary (Q) signals, are filtered and sampled to digital representation by the Analog to Digital Converters (ADC), and transformed to into frequency-domain representation by the FFT operation. This reproduces the parallel streams, $Y_i$, of subcarriers with modulated data. Finally data, $\hat{s}[n]$, is obtained after a symbol detector is used to perform the inverse operation of the constellation mapper.

![Figure 3 – OFDM receiver [2]](image)
2.2 Benefits
Using OFDM provides high spectrum efficiency, i.e. the number of bits per second that can be transmitted per Hz of bandwidth, and robustness of ISI. A combination of OFDM and an advanced coding technique results in a signal that is easy to separate from a noisy channel. It is also possible to change the up and down speed by allocating a various number subcarriers for downlink and uplink, or even to different users at the same time (OFDMA).

2.3 Disadvantages
Since OFDM uses orthogonal frequencies the frequency synchronization between the receiver and the transmitter must be very accurate. Otherwise the subcarriers will not remain orthogonal and will interfere with each other resulting in a great loss of performance. Also the timing synchronization is, despite long symbol duration, important to keep accurate to avoid ISI. The disadvantages will be compensated for and the reader is referred to chapter 7 for a detailed description of how it is accomplished.

2.4 Usage
OFDM is used in several communication standards. ADSL is the most well known wired application where high speed connections are established in existing telephone copper lines. OFDM is also used in so called HomePlug devices to establish an Ethernet connection in the power wiring network in a house or an apartment. Wireless applications using OFDM are, among others, WLAN, WiMAX and Digital Video Broadcasting – Terrestrial (DVB-T).
3 WiMAX

WiMAX, acronym for Worldwide Interoperability for Microwave Access, is a certification mark used for products based on the IEEE 802.16 family of standards, which specifies a wireless metropolitan-area network (wireless MAN) technology. The technology is intended to provide a wireless alternative to the cable modem, digital subscriber lines (DSL) or T1 connections. [3].

3.1 WiMAX as a wireless solution for fixed broadband internet access

Wireless broadband access is set up like cellular systems, using base stations (BS) that serve a radius up to several kilometers. Businesses, residences or hot spots can be connected to the base station by a subscriber station (SS). Depending on the mobility of the SS the system is referred to as either fixed or mobile. A fixed system means that a SS is a fixed access point within the network. An example is presented in Figure 4, where internet access is provided to a customer premise via a base station that distributes the signal to an outdoor mounted subscriber station. The signal is then routed from the SS via standard Ethernet cable directly to a computer or to an IEEE 802.11 hot spot to provide the end user with internet access. In a mobile system, the SS can be a mobile terminal, such as a laptop. [4].

A transmission from the BS to a SS is referred to as downlink, and a transmission from a SS to the BS is denoted uplink. WiMAX uses a scheduling algorithm to provide each SS with a time slot during which it is allowed to communicate with the BS. When a time slot has been assigned to a specific SS no other subscriber is allowed to use it.
3.2 IEEE 802.16d
The IEEE 802.16d standard is based on the OFDM modulation transmission technique and uses 256 subcarriers. The system can be configured to use any bandwidth from 1.25 to 20 MHz which implies that the subcarriers are very closely spaced. As mentioned before, closely spaced subcarriers are equivalent to larger OFDM symbol period. The closely spaced subcarriers and long symbols is the key differentiator between WiMAX systems and wireless local area networks (wireless LAN), which provides WiMAX with significant advantages for transmission over large areas and non-line-of-sight (NLOS) applications. [5].

3.2.1 Frequency domain
Each OFDM symbol is made up from 256 subcarriers and there are three types of subcarriers used for different purposes, namely

- 192 data subcarriers – used for data transmission.
- 8 pilot subcarriers – used for various estimation purposes.
- 56 null subcarriers – used for guard bands and DC carrier.

Figure 5 illustrates the frequency-domain description of an OFDM symbol. [6].

![Figure 5 – OFDM symbol frequency description](image)

The purpose of the guard bands is to let the signal to naturally decay in the frequency domain, i.e. the outermost subcarriers (closest to the guard bands) sinc-spectra’s are allowed to decay with respect to amplitude within the OFDM symbol bandwidth.

The pilot subcarriers are modulated with pre-defined data and are transmitted with higher power to enable for estimation tasks in the receiver.
3.2.2 Time domain

The time-domain symbol period, of duration $T_s$, is achieved after an inverse Fourier transformation of the frequency domain OFDM symbol, and is presented in Figure 6. [6].

![Figure 6 – OFDM symbol time structure](image)

The time-domain symbol is preceded with a cyclic prefix (CP) of duration $T_g$, which is a repetition from the end of the active symbol period ($T_b$). A longer symbol period is achieved and reduces ISI as the symbol duration becomes greater than the channel impulse response (CIR). Furthermore, the use of a CP means that a time-window of length equal to the active symbol period ($T_b$) can vary its position by as much as CP and still recover the complete symbol without ISI. The timing offset that arises due to this can be taken care of later with signal processing in the frequency-domain. [7].

The transmission of a signal consisting of multiple subcarriers can create inter-channel interference (ICI). To avoid ICI, the subcarrier frequencies are spaced by the inverse of the active OFDM symbol period to achieve orthogonality.

3.2.3 Preamble

A preamble is a predefined structure of either one or two consecutive OFDM symbols and they are used for various estimation and synchronization issues between the BS and the SS.

For downlink and network entry, the preamble shown in Figure 7 is used. The time-domain representation of the first OFDM symbol consists of a CP and four repetitions of 64-sample fragment, as a result from that only subcarriers with frequency offset indices which are a multiple of 4 are utilized. In the same manner, the second OFDM symbol consists of a CP and two repetitions of 128-sample fragment since it only utilizes even subcarriers. [6].

![Figure 7 – DL and network entry preamble structure](image)

For uplink, only the second OFDM symbol in Figure 7 is used as preamble, and it is referred to as $P_{EVEN}$. 


4 Reference model
The reference model is a floating point Matlab model of an IEEE 802.16d system. The model includes a transmitter, a channel and a receiver, and is illustrated in Figure 8.

![Reference model diagram](image)

The system model support both uplink and downlink transmission, but the thesis focus on the receiver part for uplink transmission, i.e. the receiver is assumed to be physically located inside the base station.

4.1 Transmitter
The transmitted OFDM signal is constructed in the frequency-domain by mapping subcarriers with modulated data. An inverse FFT is used to achieve a time-domain signal that can be transmitted over a radio channel.

Figure 9 shows how the transmitted signal is obtained and the functions of the sub blocks are briefly described below.

![Transmitter diagram](image)

The randomizer is used to pseudo-randomly scramble input data to avoid transmission of long sequences of the bits of the same sense. The Forward Error Correction (FEC) block adds redundancy to the transmitted information which allows the receiver to detect, and to some extent, correct errors that have occurred during the transmission. The interleaver is used to protect the transmission against burst errors. The modulator maps data into constellation points and the subcarrier mapper modulates the allocated subcarriers. The preamble generator generates a preamble to precede the burst depending on the transmission type. The pilot subcarrier mapper inserts pilot subcarriers into each data burst. The inverse FFT transforms the frequency-domain
signal into a time-domain signal and a cyclic prefix (CP) is inserted to obtain the complete OFDM signal.

### 4.2 Channel

The channel can be setup to add different signal paths, i.e. simulate a multi-path channel. It also adds a delay and white Gaussian noise with respect to selected signal to noise ratio (SNR) and signal power. Furthermore, the channel model adds errors to the signal that naturally would occur due to physical imperfection of the transmitter and receiver.

### 4.3 Receiver

The receiver performs the same operations as the transmitter, but inversed and in a reversed order. It also includes operations for synchronization and compensation for the destructive channel. These extra operations are the main focus and they will be presented and explained throughout the thesis. All signal processing is completed in the frequency-domain and the essential block of the receiver is the FFT. A simplified overview of the receiver is seen in Figure 10.

![Figure 10 – Receiver](image)

### 4.4 System parameters

The reference model specifies a number of parameters that can be found in Table 1.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( BW ) – nominal channel bandwidth</td>
<td>3.5 [MHz]</td>
</tr>
<tr>
<td>( N_{used} ) – number of used subcarriers</td>
<td>200</td>
</tr>
<tr>
<td>( n ) – sampling factor</td>
<td>8/7</td>
</tr>
<tr>
<td>( G ) – ratio of CP time to useful symbol time</td>
<td>1/8</td>
</tr>
<tr>
<td>( F_s ) – sampling frequency</td>
<td>4 [MHz]</td>
</tr>
<tr>
<td>( \Delta f ) – subcarrier spacing</td>
<td>15.6 [kHz]</td>
</tr>
<tr>
<td>( T_b ) – useful symbol time</td>
<td>64 [µs]</td>
</tr>
<tr>
<td>( T_g ) – Cyclic prefix time</td>
<td>8 [µs]</td>
</tr>
<tr>
<td>( T_s ) – OFDM symbol time</td>
<td>72 [µs]</td>
</tr>
</tbody>
</table>

Table 1 – Specified system parameters

### 4.5 Quantization

The signal to quantization ratio, SQR, is used as a measure between the result from LeoCore and the result from the reference model. The SQR is calculated according to Eq. 4-1, where \( X_{Matlab} \) and \( X_{LeoCore} \) denote the results of an operation in the reference model and LeoCore, respectively. Hence, the equation relates the quantization noise generated in LeoCore (\( X_{Matlab} - X_{LeoCore} \)) with the floating point signal in Matlab.
\[
SQR = 20 \cdot \log_{10} \frac{\sum |X_{\text{Matlab}}|}{\sum |X_{\text{Matlab}} - X_{\text{processor}}|} \quad [dB]
\] 

Eq. 4-1

A given SQR corresponds to a certain number of correct bits of the result from LeoCore. A quantization of a floating point number into \( M \) bits results in an SQR, according to Eq. 4-2.

\[
SQR = 20 \cdot \log_{10} 2^M \quad [dB]
\] 

Eq. 4-2

Table 2 shows the resulting SQR for different number of correct bits \( M \).

<table>
<thead>
<tr>
<th>Correct bits</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQR [dB]</td>
<td>24</td>
<td>30</td>
<td>36</td>
<td>42</td>
<td>48</td>
<td>54</td>
<td>60</td>
<td>66</td>
<td>72</td>
</tr>
</tbody>
</table>

Table 2 – SQR and corresponding number of correct bits
5 LeoCore technology

The processor used in this thesis is the LeoCore\(^1\) programmable baseband processor developed by Coresonic and based on research at Linköping University. The research behind LeoCore was driven by today’s need of baseband processing for SDR, as the market demands smaller mobile devices that consumes less power while managing many different radio standards. The aim was to design a processor that could support a large number of current and future radio standards and be able to dynamically adapt bandwidth and mobility via a firmware upgrade. The technology combines flexibility and performance with power consumption not much higher than for an ASIC solution. [8], [9].

The architecture is based on an application specific DSP processor with a specialized dual complex multiply-accumulate unit. The instruction set is optimized for execution of common baseband processing operations.

5.1 Single Instruction Multiple Task – SIMT

The key innovation is a new parallel architecture called Single Instruction Multiple Task (SIMT), where the principle is to let one single instruction launch multiple parallel instruction tasks. SIMT offers the performance and flexibility of state-of-the-art VLIW-SIMD solutions but with less control overhead, lower memory cost, and smaller code size. This is enabled by the optimized instruction set, an on-chip netork, distributed-addressed data memories and fixed function accelerators. The principle view of the LeoCore technology can be seen in Figure 11.

\[\text{Figure 11 – Principle view of LeoCore tecnology}\]

\(^1\) LeoCore is a trademark of Coresonic
Memories, accelerators and the processor core are connected by an on-chip network. The network is a crossbar network with configurable connections under program control that enables for multiple parallel data transfers. The accelerators are used to perform key operations that can run in parallel with the core, and they can synchronize and communicate with each other without interference from the processor. An example of how the accelerators can be chained for a WLAN application is presented in Figure 12.

![Network chain in LeoCore](image)

Accelerator chaining give raise to pipelining on symbol level – the first symbol is in the accelerator chain, the second is being processed by the core and the third is being received and stored into an available data memory. This will increase the throughput and decrease the computation demands since several operations can be preformed in parallel.

### 5.2 Processor core

The DSP core consists of one dual complex multiply accumulate unit (CMAC) and one arithmetic logic unit (ALU).

#### 5.2.1 CMAC

The CMAC is a two-way unit where the two parallel paths are used to process complex data. Each of the two paths includes one 12-bit complex multiplier, one 32-bit complex adder, one 16-bit complex adder and two 32-bit complex accumulators. The CMAC is capable of executing vector instructions and normally executes an operation on a size $N$ vector in $N/2$ clock cycles. The CMAC has three 64-bit ports to which the different memories/accelerators can be connected. The accumulators can also be loaded with data from the general registers in the ALU via a 16-bit port.

#### 5.2.2 ALU

The ALU consists of an arithmetic unit, a logic/bit manipulation unit, a barrel shifter and 16 general purpose registers. The ALU does not support complex instructions and all parts are 16-bits wide. The ALU is intended for arithmetic and control oriented tasks.
5.3 **Memories**

There are four types of memories – four data memories (DM), one coefficient memory (CM), one integer memory (IM) and one program memory (PM). The four DMs and the CM can store 32-bit complex data, i.e. 16-bit representation for the real and imaginary parts respectively. CM is connected directly to one of the three ports of the core, and the core can have at most two DM simultaneously connected. The CM is intended to store FFT twiddle factors, filter coefficients and other coefficient not to be processed by the accelerators. The IM is used to store 16-bits real data and is connected to the network. It can optionally be used as a software stack. The PM holds the machine code produced by the assembler. The five complex memories can read and write two consecutive complex samples at once, since they consists of two interleaved memory banks. This means that two even or two uneven addresses can not be read concurrently, which sometimes is needed by the FFT butterfly operation. This is solved by an FFT addressing mode that by no visible effect for the user rearranges the read/write addresses to avoid memory bank conflicts. These memories also support bit reversed addressing that is used in during the FFT calculation.

5.4 **Instruction set**

The instruction set includes ordinary classes like move instructions, shift instructions, program flow instructions (jumps and loops) and complex instructions like add and sub. It also contains network and accelerator configuration instructions, and a special class called vector instructions. All non-vector instructions are single cycle and multicycle vector instructions on the CMAC unit will execute in parallel with ALU instructions.

5.4.1 **Vector instructions**

The vector instructions operate on vectors of complex numbers stored in the memories, i.e. distributed vector addressing. The result is automatically stored to another memory unless the result is a scalar, e.g. the result of a max search. Vector instructions take, depending on vector size, multiple cycles to complete. However, instructions not using the CMAC can execute in parallel with the ongoing vector instruction. The leads to that instructions used for control overhead can be “hidden” behind the multicycle vector instruction.

Since the instruction set is optimized for baseband processing, it includes, among many others, instructions for calculation of maximum square absolute value and position of a complex vector, radix-2 FFT butterflies, sum of absolute values, vector elementwise absolute squares, etc. These instructions are important to reduce the cycle cost and facilitate the assembly programming.

One example of how part of an assembly program using vector instructions can look like is presented in Figure 13.
Part 1 in the assembly program is for setup of the network. The nwc command will setup a one way network connection from the first operand to the second. Port0-2 denotes the three ports of the CMAC.

In part 2 the memories are set up by an accelerator (acl) instruction. This instruction is used for set up of read/write addresses, address increments and other accelerator functions.

Part 3 is the FFT instruction performed on 256 data samples read from port2, and using the twiddle factors from port0. The result is stored to the port left out in the instruction, in this case port1.

In part 4 instructions not using the CMAC and independent of the ongoing calculation can be preformed in parallel with the FFT operation. When there are no more instructions that fulfill these demands an idle instruction is inserted (Part 5). The idle instruction holds the program flow until the ongoing CMAC instruction is completed.
5.5 Coresonic Developer Studio

Coresonic Developer Studio (CDS) is the development platform used throughout the thesis. It includes a cycle-true and bit-true simulator as well as assembler and debugger. The CDS environment is presented in Figure 14 and provides the user with supportive information when stepping through the assembly program.

![Coresonic Developer Studio development platform](image)

Figure 14 – Coresonic Developer Studio development platform
6 Angle calculations in hardware – CORDIC

A common task in signal processing is rotation or angle calculation of a vector. In LeoCore, it is not possible to perform trigonometric functions and an alternative way to calculate the angle must be used. The CORDIC algorithm, acronym for COordinate Rotation DIgital Computer, provides an efficient iterative method of performing vector rotations by arbitrary angles using only shifts and additions. The algorithm is derived from the general equations for vector rotation in the Cartesian coordinate system, according to Figure 15. [10].

![Figure 15 – Vector rotation in the Cartesian coordinate system](image)

The components of vector V’ are obtained from

\[
x' = x \cdot \cos(\theta) - y \cdot \sin(\theta)
\]
\[
y' = y \cdot \cos(\theta) + x \cdot \sin(\theta)
\]

Eq. 6-1

and the equations can be rewritten as

\[
x' = \cos(\theta) \cdot [x - y \cdot \tan(\theta)]
\]
\[
y' = \cos(\theta) \cdot [y + x \cdot \tan(\theta)]
\]

Eq. 6-2

The multiplication with \( \tan(\theta) \) can be avoided if the rotation angles are restricted so that \( \tan(\theta) = 2^{-i} \) for iteration \( i \), which represents a simple shift operation in hardware. With \( \theta = \arctan(2^{-i}) \) the cosine term is constant for a fixed number of iterations and can be neglected if only the angle of the final vector is of interest. The resulting equations becomes

\[
x'_{i+1} = x_i - y_i \cdot d_i \cdot 2^{-i}
\]
\[
y'_{i+1} = y_i + x_i \cdot d_i \cdot 2^{-i}
\]

Eq. 6-3

The direction of each rotation is defined by \( d_i \) and the sequence of all \( d_i \)'s determines the final vector. Depending on how \( d_i \) is defined, the CORDIC rotator is operated in two different modes.
One is called rotation mode and the other is called vectoring mode. Here, only the latter is discussed where the value of \( d_i \) is determined by the \( y_i \) component according to

\[
d_i = +1, \quad \text{if } y_i < 0
\]
\[
-1, \quad \text{if } y_i \geq 0
\]

A third equation is added that accumulates the rotated angles at each iteration.

\[
\varphi_{i+1} = \varphi_i - d_i \cdot \arctan(2^{-i})
\]

Eq. 6-4

The values of \( \arctan(2^{-i}) \) are pre-calculated and stored in a look-up table and accessed once per iteration.

The interpretation of the CORDIC equations for vectoring mode is:

An input vector is rotated through whatever angle is necessary to align the resulting vector with the x axis. This is done by trying to minimize the y component of the residual vector at each rotation. The sign of the y component determines which way to rotate the vector next. The accumulator will contain the traversed angle at the end of the iterations. For the given algorithm the rotation angle is limited to angles between \( -\pi/2 \) and \( \pi/2 \) due to the tangent function argument in the first iteration. However, extending the rotation angles can be done by mirroring the input vector to the first quadrant while recording its origin.

### 6.1 Implementation

A flowchart of the implementation is presented in Figure 16. The input vector is assumed to be the result of a preceding operation in the CMAC unit and the real and imaginary parts of the accumulator are moved to two 16-bit registers representing the x- and y-components. The input vector is then mirrored to the first quadrant and if \( x < 0 \) the vector origin in the left half plane (\( m_x = 1 \)) and if \( y < 0 \) it also origin in the bottom half plane (\( m_y = 1 \)). An enhancement to the algorithm is to exploit that if the y component is larger than the x component, the vector can be mirrored in the angle \( \pi/4 \) (\( m_{45} = 1 \)). Then the CORDIC table can be started at \( \arctan(2^{-1}) \). After the iterations are performed the angle of the vector is stored in a 16-bit register and finally the correct angle is determined within the interval \( [-\pi, \pi] \) by considering the origin of the vector.

The IM of the processor holds the CORDIC table as shown in Table 3 and should already be copied from the CM at the startup of the processor. The number of iterations is decided to be twelve. Further iterations would not necessary give a more correct angle since the maximum error in the angle from rounding of the arctangent values is 0.014° which is larger than \( \arctan(2^{-13}) = 0.0070° \). Since angle calculations appear frequently in signal processing, the CORDIC algorithm is implemented as a subroutine, hence the assembly code of the angle calculation only needs to be included once in the main program.
Figure 16 – CORDIC flowchart
\[
\begin{array}{|c|c|c|}
\hline
i & \text{round}(\arctan(2^i) \cdot 32768/\pi) & \arctan(2^i)^{\circ} \\
\hline
1 & 4836 & 26.5650 \\
2 & 2555 & 14.0362 \\
3 & 1297 & 7.1250 \\
4 & 651 & 3.5763 \\
5 & 326 & 1.7899 \\
6 & 163 & 0.8952 \\
7 & 81 & 0.4476 \\
8 & 41 & 0.2238 \\
9 & 20 & 0.1119 \\
10 & 10 & 0.0559 \\
11 & 5 & 0.0280 \\
12 & 3 & 0.0140 \\
\hline
\end{array}
\]

Table 3 – CORDIC table of arctangent values stored in IM

The unit circle has been shared into uniformly distributed points instead of using radians or degrees, i.e. the result from the CORDIC subroutine should be interpreted as the angle (in radians) divided with pi on the interval \([-2^{15}, 2^{15})\). The profit with this representation is twofold. First, the angle register can be used as memory read address for a fixed size look-up table stored in CM. Secondly, additions of several angles or multiplications between an angle and an integer will never lead to an overflow since the processor only used two’s complement representation. The angle representation is presented in Figure 17.

For example, if the two positive angles \(\pi/3\) and \(5\pi/6\) are added, the result should be \(7\pi/6\). The 16-bit binary result from the addition is:

\[
\begin{align*}
0.110'1010'1010'1010 & = \frac{5\pi}{6} \\
0.010'1010'1010'1010 & = \frac{\pi}{3} \\
1.001'0101'0101'0100 & = -\frac{5\pi}{6}
\end{align*}
\]

As seen, an overflow has occurred since the result is negative (after adding two positive numbers). However, this angle, \(-5\pi/6\), is the same as \(7\pi/6\) on the unit circle and the result is thus correct.
6.1.1.1 Performance
The complete algorithm is implemented with 80 lines of assembly code and requires 176 clock cycles to complete.
7 Baseband processing

This chapter explains the purpose of the baseband processing required in the receiver. An introduction to the theory behind the reference model algorithms is presented first in each subsection, followed by a description of the implementation in LeoCore. The baseband processing in the receiver, presented in Figure 18, consists of packet detection, FFT, channel estimation, phase tracking and compensation, and demodulation.

7.1 Packet detection

To be able to allow new subscriber stations into the network there is a contention slot for initial ranging in the IEEE 802.16d frame structure. Within this time slot it is free for any new SS to send a request to the BS [6]. The BS needs to determine if there is a request or not within the contention slot and this is called packet detection. The reference model uses the structure of the 4x64 preamble sent in the beginning of every request for determination. Autocorrelation of 3x64 samples between the received signal and a delayed version with 64 samples will give rise to a step when the preamble appears. This is illustrated in Figure 19. To further secure the appearance of the preamble the power estimate at that certain moment is compared with a threshold value. If both these requirements are fulfilled at the same time, preamble detection is considered. These values are recalculated for every new sample and the whole preamble has to be received before the start can be detected. This will put unrealistic high demands on the hardware so a compromise is needed.

![Figure 18 – Baseband processing in the receiver](image-url)
LeoCore has a Digital Front End (DFE) that among other things handles the packet detection. The packet detector uses a window of 16 samples, instead of 3x64 as in the reference model, and generates an interrupt to activate the core when the preamble is present. This will not give the same accuracy as the reference model since a smaller sliding window may miss the presence of a packet or possibly indicate a false detection. However, the task of packet detection is just a rough approximation of the packet start since the exact value is calculated using the second preamble in the core. When the approximate start of the preamble is detected the DFE will wait until the first preamble ends (since that is not further used) and then start sending the second OFDM symbol in the preamble to one of the data memories in the processor. One advantage of using an accelerator that handles the packet detection is that the core can enter an idle mode while the DFE handles the packet detection by itself and thereby save a lot of power. This is more important in a mobile unit where the power supply (battery) is limited.
7.2 The Discrete Fourier Transform – DFT

An OFDM system performs signal processing in the frequency-domain. To transform the time-domain signal to frequency-domain representation, the Discrete Fourier Transform (DFT) is used. The DFT for a discrete-time signal of length N, \( x(n) \), is an invertible, linear transformation defined as

\[
X(k) \equiv \frac{1}{\sqrt{N}} \cdot \sum_{n=0}^{N-1} x(n) \cdot W^{nk}, \quad W = e^{-j\frac{2\pi n}{N}}, \quad k = 0, 1, \ldots, N - 1 \quad \text{Eq. 7-1}
\]

and \( X(k) \) is a periodic, complex sequence also of length N. \[11\].

The inverse DFT, IDFT, is

\[
x(n) = \frac{1}{\sqrt{N}} \cdot \sum_{k=0}^{N-1} X(k) \cdot W^{-nk}, \quad n = 0, 1, \ldots, N - 1 \quad \text{Eq. 7-2}
\]

The normalization factor multiplying the DFT and IDFT, and the signs of the exponent are merely conventions and may be changed. The requirements of these conventions are that the DFT and IDFT must have opposite sign exponents and that the product of the normalization factors must be \( 1/N \).

Recall the discrete-time Fourier transform, DTFT, which is a function of a continuous frequency \( \omega T \in [\pi, \pi] \), while the DFT is a function of discrete frequency \( \omega_k \). The discrete frequencies \( \omega_k = 2 \cdot \pi \cdot k / N \) are given by the angles of \( N \) points uniformly distributed along the unit circle in the complex plane. Furthermore, the DFT is a sampled version of the DTFT which makes the DFT more suitable for digital implementation.

7.2.1 DFT implementation – FFT

A direct computation of the DFT requires \( O(N^2) \) arithmetical operations including complex multiplications which are rather time-consuming. The Fast Fourier Transform, FFT, denotes a class of algorithms for efficient computation of the DFT and its inverse by \( O(N \cdot \log_2(N)) \) operations. For large values of \( N \), the difference in execution time is very large between direct computation of the DFT and an FFT implementation. \[12\].
7.2.2 Radix-2 Sande-Tukey FFT algorithm

The Sande-Tukey algorithm is based on a divide-and-conquer approach in the frequency domain and is therefore referred to as decimation-in-frequency (DIF) FFT. To derive the algorithm, the DFT formula is split into two summations

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

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

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

Eq. 7-3

Now, since \(W_N^{N/2} = (-1)^k\) we get

\[
X(k) = \sum_{n=0}^{N-1} \left( x(n) + (-1)^k \cdot x(n + \frac{N}{2}) \right) \cdot W_N^{nk}
\]

Eq. 7-4

\(X(k)\) can be split (decimate) into even- and odd-indexed frequency samples:

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

Eq. 7-5

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

Eq. 7-6

The procedure can be repeated through decimation of the \(N/2\)-point DFTs \(X(2k)\) and \(X(2k+1)\), where the process involves \(\log_2(N)\) stages. The computation of the \(N\)-point DFT via decimation-in-frequency FFT requires \(N/2 \cdot \log_2(N)\) complex multiplications and \(N \cdot \log_2(N)\) complex additions. [12].
As an illustration, the signal flow graph for an 8-point decimation-in-frequency FFT algorithm is shown in Figure 20.

![Figure 20 – 8-point decimation-in-frequency FFT signal flow graph](image)

The basic operation in the signal flow graph is the butterfly operation, shown in Figure 21, which consists of two branches with operations indicated in the figure. The lower branch, $B$, contains multiplication with the twiddle factor, $W_p^N$.

![Figure 21 – Decimation-in-frequency butterfly operation](image)
7.2.3 Implementation

The vector instruction FFT.n calculates one stage, containing N/2 butterflies, of a complete N-point FFT. The input data and intermediate data storage must use FFT-addressing mode to avoid memory bank conflicts. Also, data must be moved between a memory and the CMAC with bit reversed ordering of the memory addresses with respect to current stage of the FFT. The FFT butterfly takes two samples and, as seen in Figure 20, the samples will not be adjacent in all layers. Since the processor always reads neighboring addresses the two desired addresses should be placed together. The addresses are virtually moved to adjacent addresses by reversing some of the bits of the address pointers so that the CMAC unit reads from two addresses located at different parts of the memory. The two resulting samples will be written to the same address they are read from, except in the last layer where all bits but one must be reversed due to the algorithm seen in Figure 20.

7.2.3.1 Design flow

The design flow of how to implement the FFT is as follows.

- Setup network connections
- Setup CM
  - point to twiddle factors
  - modulo addressing
  - read address
  - read step
- Setup memory read
  - read address
  - FFT addressing mode
  - bit reversal
- Setup memory write
  - write address
  - FFT addressing mode
  - bit reversal
- 1\textsuperscript{st}, 2\textsuperscript{nd}, 3\textsuperscript{rd}, …, 6\textsuperscript{th} layer butterflies
- Swap memories before next layer butterflies
- Increase CM read step before next layer butterflies
- Update bit reversal for both memories
- 7\textsuperscript{th} layer butterflies
- Swap memories before next layer butterflies
- Increase CM read step before next layer butterflies
- No bit reversal for read memory
- Bit reversal for write memory
- 8\textsuperscript{th} layer butterflies
### 7.2.4 Twiddle factors in CM

The twiddle factors are pre-calculated and stored, layer by layer, in the CM. Since the CM uses double read addressing, where only the latter value is used in the butterfly operation, the last coefficient for each layer is inserted at address zero, followed by the rest of them. Using modulo addressing will provide the value stored at address zero as the last value. In the second layer the same coefficient will be used twice, hence 64 values are stored and modulo addressing is used to wrap around. For the third layer, 32 coefficients are used four times, for the fourth layer 16 coefficients are used eight times and so on. The twiddle factors must be stored in a bit reversed order so that correct data with corresponding twiddle factor is read during each butterfly operation. Table 4 shows the twiddle factors stored in CM. To avoid memory bank conflicts, the coefficients for layer five, six and seven are placed within one memory section. In the last layer all coefficients are the same, namely $e^{0} = 1$ which is the same as the first coefficient in layer 1. Hence, prior to the last layer, the CM read address pointer and address increment is set to zero to repeatedly read the same value for all butterflies.

<table>
<thead>
<tr>
<th>Address (size)</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-127 (128)</td>
<td>Layer 1</td>
</tr>
<tr>
<td>128-191 (64)</td>
<td>Layer 2</td>
</tr>
<tr>
<td>192-223 (32)</td>
<td>Layer 3</td>
</tr>
<tr>
<td>224-239 (16)</td>
<td>Layer 4</td>
</tr>
<tr>
<td>240-255 (8+4+2)</td>
<td>Layer 5-7</td>
</tr>
</tbody>
</table>

Table 4 – Twiddle factors in CM

### 7.2.4.1 Performance

The total number of operations needed for a 256 points DIF FFT is 3072, i.e. 2048 additions and 1024 multiplications. The total cycle cost for the implemented FFT algorithm is 1117 clock cycles, meaning that LeoCore performs approximately three operations per clock cycle. The total amount of assembly instructions is only 109. This is an example of the strength in the SIMT technology used in LeoCore. The SQR after the FFT will be approximately 46 dB.

In total 3 DMs are used for the FFT calculation, 256 addresses are used in the CM for twiddle factors and 256 addresses are used in two DMs resulting in a total memory usage of 768 complex memory cells.
7.3 Channel Estimation

Channel estimation is the task of estimating the frequency response of the radio channel that the transmitted signal travels before it reaches the receiver antenna. [13].

Channel estimation is performed on the data preamble ($P_{EVEN}$) transmitted first in uplink burst. $P_{EVEN}$ consists of one OFDM symbol utilizing only even subcarriers for data transmission, that is frequency offset indices {-100, -98, ..., -2, 2, 4, ..., 100}.

Figure 22 presents an overview of how the reference model performs channel estimation. The different blocks will be described throughout this chapter.

![Figure 22 - Channel estimation tasks](image)

The initial channel estimate is obtained by dividing the received preamble samples, $R_{Peven}(k)$, by the transmitted, i.e. known preamble samples, $T_{Peven}(k)$. This result in an initial least-square, LS, channel estimate for all used subcarriers and the LS-estimated channel frequency response samples are given by

$$H(k) = \begin{cases} \frac{R_{Peven}(k)}{T_{Peven}(k)}, & k \in \{-100, -98, -..., -2, 2, 4, ..., 100\} \\ 0 & \text{else} \end{cases}$$

Eq. 7-7

The subcarriers of the received symbol are rotated with an angle that depends on the subcarrier index $k$ resulting in that the outermost subcarriers are rotated the most. The rotation of the subcarriers is a result of that the received OFDM symbol samples will drift in time due a sampling clock frequency offset, SFO, between the oscillators used in the transmitters DAC and the receivers ADC.
Since the subcarriers are rotated with respect to their subcarrier indices, this can be observed as a phase ramp among the subcarriers. The phase ramp is determined by multiplying the conjugated channel frequency response sample on the leftmost subcarrier with the adjacent sample, and repeating the procedure over all used subcarriers going from negative subcarrier indices to positive. The result is a set of phasors that are added together resulting in a phase ramp. Taking the angle of the phase ramp gives a measure of the rotation from one subcarrier to its neighbor. The symbol timing offset is calculated and compensated for in the frequency domain by exploiting the Fourier transform relation between a symbol time displacement and a phase growing linearly with frequency, i.e. a phase ramp.

When the symbol timing offset is compensated for, all subcarriers used for guard bands are filled up with appropriate channel frequency response sample i.e. subcarriers used for the left guard band obtain the value of the sample of the leftmost used subcarrier and, in the same manner, the subcarriers used for the right guard band obtain the sample of the rightmost used subcarrier. The DC subcarrier is given the value of the first positive subcarrier. This is illustrated in Figure 23.

![Figure 23 – Channel frequency response samples for DC- and guard band subcarriers](image)

When all even subcarriers have obtained channel frequency response samples, the channel estimate is filtered\(^2\). This leads to interpolated frequency response samples for odd indices of \(k\) and the filtering will suppress the influence of the noise.

The channel estimate is required in the phase tracking algorithm of the pilot tones and in the demodulation of the received OFDM symbols.

### 7.3.1 Implementation

#### 7.3.1.1 LS-estimated frequency response samples

Calculating the channel frequency response samples, according to Eq. 7-7, involves division which is not supported in LeoCore. Instead, the division is performed as a multiplication by the received samples and the pre-calculated values of one over the known samples. Totally 100 of these multiplications need to be computed as well as storing zeros on all odd subcarrier indices.

---

\(^2\) The filter operates in the frequency-domain.
This is solved by storing zeros in between the pre-calculated samples in the CM, and then using a vector instruction of length 200 (mul.200). The frequency response for all even subcarriers is then obtained and the zeros are automatically placed on odd subcarriers.

The LS-estimation of the channel estimate is done in 30 lines of assembly code and requires 128 clock cycles to complete.

**7.3.1.2 Phase ramp calculation**

The phase ramp is calculated as

\[
\text{Phase ramp} = \sum_{k=0}^{98} \text{conj}\{H(k)\} \cdot H(k + 2)
\]

Eq. 7-8

To calculate the frequency domain phase ramp, which is a sum of several multiplications, the channel frequency response samples need to be copied to a different DM. The copy is performed by the vector instruction for vector move, vmove.n, which moves n data from one memory to another. By doing this, the assembly instruction for multiply-accumulate, mac.n, which multiplies data from two different memories and accumulates the products can be used. Since every other sample is a zero a mac.200 must be performed. The memory read pointers are set up to address samples with one sample displacement and the conjugation is performed by enabling the conjugate flag for the CMAC.

Calculating the phase ramp is done in 48 lines of assembly code and requires 272 clock cycles to complete.

The angle of the frequency domain phase ramp, denoted \(dPdSC\), is calculated by calling the CORDIC subroutine and the calculation is done in 80 lines of assembly code and requires 192 clock cycles to complete.

**7.3.1.3 Symbol timing offset**

The symbol timing offset, denoted \(dkP\), is calculated as

\[
dkP = \text{round}(\text{constant} - \frac{N_{\text{FFT}}}{2 \cdot \pi} \cdot dPdSC) - \text{aim} = \text{round}(8 - 128 \cdot \frac{dPdSC}{\pi}) - 8
\]

Eq. 7-9

Since \(dPdSC\) is an angle within the interval \([-\pi, \pi]\) it can be observed that \(dkP\) is a number within the interval \([-128, 128]\).

The implementation exploits that \(dPdSC/\pi\) is a number on the interval \([-1,1]\). The division is never performed because angles in the processor are interpreted as the angle divided with pi.

After the multiplication with 128, the number is within the interval \([-128,128]\) but the actual multiplication is never performed. Instead, the 8 in Eq. 7-9 can be represented in a register with 16-bits fixed point representation according to (0000 1000 . 0000 0000)₂, that is, \(dPdSC/\pi\) is interpreted as a 16-bit number within the interval \([-128,128]\) and subtract it from an “8”
represented with the upper eight bits of a 16-bit register. Rounding is performed by adding 0.5 represented in the last mentioned interval, i.e. (0000 0000 . 1000 0000)\textsubscript{2} is added. Then an “8” is subtracted from the rounded result and the correct result is obtained after it is moved to the lower eight bits of the 16-bit register using arithmetic shifts. Now \(dkP\) is represented as a fixed point number according to \((xxxx xxxx xxxx xxxx)\textsubscript{2}.

With this implementation, \(dkP\) can be calculated in the ALU since multiplications are completely avoided and require only 5 clock cycles.

### 7.3.1.4 Phase ramp compensation

Once the timing offset is determined the frequency response samples needs to be compensated. This is done by de-rotating the angles of the frequency response samples with respect to both the timing offset and the subcarrier index according to

\[
H(k) = H(k) \cdot e^{\frac{2\pi}{N_{pp}} dkP}
\]

Eq. 7-10

The complex exponential function is implemented as a look-up table where the value of \(dkP\) multiplied with \(k\) is used as input value. The size of the table is minimized by using periodicity of the complex exponential function. Hence, the maximum input value to the table is 255. Furthermore, only positive table input values are used since a negative value of \(dkP\) or \(k\) only results in a complex conjugate of the complex exponential function. This is solved by conjugating the output value from the look-up table before the subsequent multiplication is performed with the frequency response. The look-up table is pre-calculated and stored in the CM, as shown is Table 5.

<table>
<thead>
<tr>
<th>(dkP \cdot oksc)</th>
<th>(e^{\frac{2\pi}{N_{pp}} dkP \cdot oksc})</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1+0j</td>
</tr>
<tr>
<td>1</td>
<td>0.9979+0.0245j</td>
</tr>
<tr>
<td>254</td>
<td>0.9988-0.0491j</td>
</tr>
<tr>
<td>255</td>
<td>0.9997-0.0245j</td>
</tr>
</tbody>
</table>

Table 5 – Complex exponential function look-up table

The multiplication of the frequency response and the complex exponential function is performed in two steps where we take advantage of the fact that \(P_{EVEN}\) only makes use of even subcarriers and that we are free to select the order of which the frequency response samples are multiplied.

First, \(dkP\) is set to its absolute value and the data memory which holds the frequency response is setup to read samples stored on address 2, 4, 6, ..., 100. A loop of length equal to half of the even subcarriers, i.e. 50, is initiated that, for each iteration:

- add \(dkP\) with twice the original value of \(dkP\), i.e. \(2 \cdot dkP\), \(4 \cdot dkP\), \(6 \cdot dkP\) etc. which corresponds to \(dkP \cdot k\) for positive values of \(k\).
- subtract 256 if the result is larger or equal to zero, i.e. make use of the periodicity of the complex exponential function. Then the resulting value is used as an address pointer to the look-up table to obtain the value of the complex exponential function.
- set the conjugate bit in the CMAC if \( dkP \) was negative, i.e. complex conjugate of the look-up table value before the multiplication with the frequency response.
- store the result of the multiplication on the same address from which the frequency response sample was taken from.

Then, the program is re-used after setting the read address of the data memory to read frequency response samples stored on address 254, 252, 250, ..., 156 and then the loop is entered once again. This time the conjugate bit for the CMAC is set only if \( dkP \) is positive since this time the values of \( k \) are negative but still only positive input values are used to the look-up table.

As a clarification of the described algorithm an example is presented. Assume that \( dkP \) is determined to -78 samples. First the absolute value of \( dkP \) is taken, i.e. \( dkP = 78 \), while remembering that \( dkP \) was negative. The first index of the positive subcarrier is \( k = 2 \), hence \( dkP \) is added with itself to \( 2 \cdot 78 = 156 \). Then 256 is subtracted, resulting in \( 156 - 256 = -100 \) which is not greater or equal to zero, hence the subtraction is undone. The value 156 is now used as to point to address 156 in the CM from which the value of the complex exponential function is obtained, that is \( e^{j \frac{2\pi \cdot 78 \cdot 2}{256}} \). Since \( dkP \) was originally negative the conjugate flag is enabled and the multiplication of frequency sample (pointed out from the data memory) with the value from the look-up table is performed.

In the next iteration of the loop, \( dkP \) should be multiplied with the preceding index value from the \( k \) vector, which is \( k = 4 \). This is done by adding the last value of \( dkP \) (= 156) with twice the original value of \( dkP \), that is \( 156 + 2 \cdot 78 = 312 \). Now, 256 is subtracted resulting in \( 312 - 256 = 56 \).

The value 56 is used as input value for the look-up table generating \( e^{j \frac{2\pi \cdot 78 \cdot 4}{256}} \). The conjugate flag is already enabled in the last iteration and the multiplication of the table value with the corresponding frequency response sample is performed.

In the same manner, iteration three yield the table input value \( 56 + 2 \cdot 78 = 212 \) with which \( e^{j \frac{2\pi \cdot 78 \cdot 6}{256}} \) is obtained.

The loop is repeated 50 times and the frequency response samples with positive indexes are compensated for, one sample per iteration. Then the algorithm is repeated for the frequency response samples with negative indexes by entering the loop a second time. Since \( dkP \) originally was negative (-78) and the subcarrier indexes now are negative (-100, -98, ..., -2) the conjugate flag is disabled prior to the multiplication between each frequency response sample and the complex exponent value.

The phase ramp compensation is done in 67 lines of assembly code and requires 1527 clock cycles to complete.
7.3.1.5 Channel estimate filtering

The frequency response of the channel estimate contains samples only on all even data subcarriers. To get a complete channel estimate, the frequency response sample with index just prior to the guard bands is copied, as illustrated in Figure 23, i.e. the sample on subcarrier index 100 is copied to samples with subcarrier indexes 102, 104, …, 126 and the sample on subcarrier index -100 is copied to samples with subcarrier indexes -102, -104, …, -126. At the same time the subcarriers with odd indices within the guard bands are set to zero. Also, the frequency response sample with subcarrier index 2 is copied to obtain a sample for the DC-subcarrier.

After that, the channel estimate is filtered so that a total channel estimate for all subcarriers is achieved.

The filter is a 20’th order lowpass FIR filter of type I (h(n) = h(20-n)) with linear phase, constructed with Matlab’s fir1 function. The filter coefficients are scaled with safe scaling to prevent overflow. The impulse response of the FIR filter is shown in Figure 24 and values of the filter taps are listed in Table 6. The magnitude response of the FIR filter is shown in Figure 25 where the cut-off frequency is 0.2 and the normalized gain of the filter at the cut-off frequency is -6 dB.
Figure 25 – Magnitude response of FIR filter

<table>
<thead>
<tr>
<th>( n )</th>
<th>( h(n) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>-0.0021</td>
</tr>
<tr>
<td>2</td>
<td>-0.0063</td>
</tr>
<tr>
<td>3</td>
<td>-0.0116</td>
</tr>
<tr>
<td>4</td>
<td>-0.0124</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0.0318</td>
</tr>
<tr>
<td>7</td>
<td>0.0814</td>
</tr>
<tr>
<td>8</td>
<td>0.1375</td>
</tr>
<tr>
<td>9</td>
<td>0.1821</td>
</tr>
<tr>
<td>10</td>
<td>0.1992</td>
</tr>
<tr>
<td>11</td>
<td>0.1821</td>
</tr>
<tr>
<td>12</td>
<td>0.1375</td>
</tr>
<tr>
<td>13</td>
<td>0.0814</td>
</tr>
<tr>
<td>14</td>
<td>0.0318</td>
</tr>
<tr>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>-0.0124</td>
</tr>
<tr>
<td>17</td>
<td>-0.0116</td>
</tr>
<tr>
<td>18</td>
<td>-0.0063</td>
</tr>
<tr>
<td>19</td>
<td>-0.0021</td>
</tr>
<tr>
<td>20</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 6 – FIR filter taps
The channel estimate is filtered through the FIR filter using Matlab’s filter function and the FIR filter structure is shown in Figure 26.

![Figure 26 – Direct form FIR filter structure](image)

The direct form FIR structure is described by the difference equation:

$$y(n) = \sum_{k=0}^{20} h(k) \cdot x(n - k)$$

\text{Eq. 7-11}

The implementation of the filter utilizes that the first and the last filter taps are zero, i.e. $h(0) = h(20) = 0$. This results in that:

- the first output sample is always zero
- calculating output samples 1 to 19 includes multiplication with one zero for each sample
- calculating output samples 20 to 255 includes multiplication with two zeros for each sample

The multiplications with zeros are time-consuming when processing 256 output samples and are avoided in the implementation. In total, the implementation avoid $1+19+236 \cdot 2 = 492$ zero multiplications.

The non-zero filter taps are stored in the coefficient memory of the processor and the data memory that holds the channel estimate is setup to read two samples at a time. For each output sample to be calculated, the processor executes the required number of multiply-and-accumulate instructions according to the difference equation above. The starting addresses of the filter taps and input data belonging to the subsequent multiply-and-accumulate instruction is calculated (at no extra cycle cost) while waiting for the current result to go through the pipeline of the processor. Each output sample that is calculated is rounded, saturated and then stored to a complex data memory.
An overview of the implementation is described below.

- **Setup**
  - Setup CM
    - point to segment where filter taps are stored
    - point to first filter tap to be used
    - initiate modulo read addressing of 19 filter taps
    - read two taps at a time
  - Setup memory holding input samples
    - point to first sample
    - initiate modulo read addressing of 256 samples
    - read two samples at a time
  - Setup memory for output samples
    - point to address where first output sample should be stored
    - initiate modulo writing of 256 samples

- **Prologue**
  - store 0 as first output sample
  - calculate output samples until the filter is filled with input samples
    - execute multiply and accumulate instructions of enumerating size
    - calculate next addresses of taps and input data for next output sample
    - round, saturate and save result

- **Filter kernel**
  - Initiate a loop of 236 iterations and calculate the remaining output samples
    - execute multiply and accumulate instructions over 19 samples
    - calculate next addresses of taps and input data for next output sample
    - round, saturate and save result

It should be pointed out that filtering the channel estimate requires
\[ 1 + 2 + \ldots + 19 + 19 \cdot 236 = 4674 \] non-zero complex multiplications and additions which normally would be extremely time-consuming. However, using the processor's vector instruction for multiplication and accumulation, we can by far reach a much lower clock cycle cost.

The filter taps are real valued but there would be no extra cost to make use of complex filter taps.

The channel estimate filtering is done in 214 lines of assembly code and requires 2850 clock cycles to complete.

### 7.3.1.6 Resource allocation

Since all operations during the process of channel estimation are sequential, the content of a data memory can be overwritten after it has been used. In total two complex data memories are used. One DM holds the frequency-domain samples of the received preamble and some intermediate results. In total 256 memory addresses are allocated within this memory. The second DM serves as a workspace and 200 memory addresses are used within that memory. The CM holds (except for the look-up table) the inverse of the transmitted frequency-domain preamble samples (100)
with intermediate zeros (99) and the filter taps (12). Also, 100 memory addresses must be used to store an intermediate result from a vector operation which reads from two data memories. In total 311 memory addresses are allocated in CM.

### 7.3.1.7 Performance

The reference model in Matlab is used to generate test data for the processor. Then channel estimation is performed with a finite number of bits in the processor and the result is compared to Matlab’s floating point channel estimate. The comparison is the SQR of the sum of all frequency response samples according to

\[
SQR_{\text{channel estimation}} = 20 \cdot \log_{10} \left( \frac{\sum_{k=0}^{255} |H_{\text{Matlab}}(k)|}{\sum_{k=0}^{255} |H_{\text{Matlab}}(k) - H_{\text{Processor}}(k)|} \right) \quad \text{Eq. 7-12}
\]

The resulting SQR for the implementation of the channel estimation is nearly 55 dB which corresponds to an accuracy of 9 correct bits out of 16 possible. Recall that the CMAC multipliers truncate data to 12 bits prior to the multiplication and therefore we can not achieve a more accurate result than 12 correct bits.

In total, the channel estimation is done in 444 lines of assembly code and requires 4974 clock cycles to complete. Table 7 shows the performance of the complete channel estimate implementation where it can be seen that the most time-consuming task is to lowpass filter the channel estimate.

<table>
<thead>
<tr>
<th>Task</th>
<th>Lines of assembly code</th>
<th>Clock cycles</th>
<th>Percent of total clock cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>LS-estimation</td>
<td>30</td>
<td>128</td>
<td>2.6</td>
</tr>
<tr>
<td>Phase ramp</td>
<td>128</td>
<td>464</td>
<td>9.3</td>
</tr>
<tr>
<td>Timing offset</td>
<td>5</td>
<td>5</td>
<td>0.1</td>
</tr>
<tr>
<td>Phase ramp compensation</td>
<td>67</td>
<td>1527</td>
<td>30.7</td>
</tr>
<tr>
<td>Filtering</td>
<td>214</td>
<td>2850</td>
<td>57.3</td>
</tr>
<tr>
<td>Complete channel estimate</td>
<td><strong>444</strong></td>
<td><strong>4974</strong></td>
<td><strong>100</strong></td>
</tr>
</tbody>
</table>

*Table 7 – Channel estimation performance*
7.4 Phase tracking

The performance of an OFDM system is very sensitive to both timing offset and frequency offset, and synchronization of such errors is crucial for the retrieval of the transmitted information. This chapter describes how phase tracking and compensation is completed in the reference model.

7.4.1 Frequency offset

The uncertainty in the carrier frequency is due to a difference in the frequencies of the local oscillators in the transmitter and receiver. \[13\].

Since frequency is defined to be the derivative of the phase with respect to time, a frequency error can be described as a cumulative phase error that linearly increases or decreases with time, depending on the sign of the frequency error. This can be seen in the following discussion:

If the baseband signal is \(x(t)\) and the carrier frequency is \(f_c\), then the transmitted signal, \(s(t)\), becomes

\[
 s(t) = e^{j2\pi f_c t} \cdot x(t) \quad \text{Eq. 7-13}
\]

resulting in a received signal, denoted \(r(t)\)

\[
 r(t) = e^{j2\pi (f_c - f'_e) t} \cdot x(t) \quad \text{Eq. 7-14}
\]

The received signal has a phase offset, denoted \(\phi(n)\), equal to

\[
 \phi(n) = 2\pi \cdot (f_c - f'_e) \cdot n \cdot T = 2\pi \cdot \Delta f \cdot n \cdot T \quad \text{Eq. 7-15}
\]

where \(\Delta f\) is the frequency offset. Hence, the phase offset is linearly increasing or decreasing with time.

7.4.2 Timing offset

The sampling clock frequency error results in a timing error because the sampling instants between the transmitter and receiver differ. \[13\].

The time-continuous Fourier transform relationship

\[
 F\{x(t - \tau)\} = e^{-j\omega \tau} \cdot X(\omega) \quad \text{Eq. 7-16}
\]

illustrates that a timing offset \(\tau\) of the received signal will give rise to a phase offset \(\omega \cdot \tau\) that grows linearly with frequency, i.e. subcarrier frequency offset index.
7.4.3 Time and frequency-domain effects

The effect from the timing and frequency offsets can be seen in Figure 27 where $k$ is the subcarrier offset index and $l$ is the OFDM symbol number, i.e. time index. The subcarriers are rotated in phase and the phase rotation is a function of time (symbol number) which is different between the subcarriers. [14].

![Figure 27 – Subcarrier phase rotation for adjacent OFDM symbols](image)

For the reader with a creative mind, the effects can be visualized by imagining the phase rotation as an airplane:

Let the origin of the coordinate system in Figure 27 be placed in the middle of the airplane so that increased phase rotation corresponds to a higher altitude of the airplane. Also, let the negative and positive frequency offset indices $k$ corresponds to the left and right airplane wings respectively. The elapsed time since the beginning of the burst is a dimension pointing into the paper but it seen as broken lines with an increasing OFDM symbol number.

The airplane will rise (or drop) to a higher altitude when more symbols are received as a result of that the frequency offset grows linearly with time depending on the sign of the frequency error.

A roll to the left (as can be imagined in Figure 27) or right of the airplane would then correspond to a timing offset which causes a phase offset that grows linearly with the subcarrier frequency offset index and as more symbols are received the roll will increase since the phase offset increases with time.
### 7.4.4 Pilot subcarriers

The receiver must estimate the phase rotation of each data subcarrier to be able to eliminate or at least minimize the impact of the frequency error. There are two main approaches for this where the first is to adjust the oscillator in order to align the receiver clock with the transmitter clock. The second approach is to use post-processing in the frequency domain [13]. This is done with the 8 pilot subcarriers submitted in each data OFDM symbol. The pilot subcarriers are BPSK modulated carrying known data that the receiver can use to perform synchronization functions.

The pilot subcarriers are transmitted on frequency offset indices

\[ k \in \{-88, -63, -38, -13, 13, 38, 63, 88\} \]

A description of the phase tracking algorithm is presented together with the implementation in the following sections.

The total phase rotation, as described above, is a linear function of time which is different between the subcarriers. Hence, all data subcarriers should be de-rotated in phase with an angle, \( P_{\text{comp}} \), according to

\[
P_{\text{comp}} = C(1) + C(2) \cdot (\text{OFDM symbol number}) + C(3) \cdot (\text{subcarrier index}) + C(4) \cdot (\text{subcarrier index}) \cdot (\text{OFDM symbol number})
\]

Eq. 7-17

In Eq. 7-17

- \( C(1) \) is the average phase difference for all subcarriers,
- \( C(2) \) is the phase difference with respect to time,
- \( C(3) \) is the phase difference between two subsequent subcarriers,
- \( C(4) \) is the phase difference due to timing drift within a burst.

In Figure 27

- \( C(1) \) corresponds to where a broken line intercepts the phase rotation axis,
- \( C(2) \) corresponds to the distance between two broken lines on the phase rotation axis,
- \( C(3) \) corresponds to the slope of a broken line.

When the values of \( C(i) \) have been determined, the frequency and timing offset can be estimated as and reported to the media access control layer. The information will be included in a downlink burst from the BS to the SS so that the local oscillator can be adjusted.
7.4.5 Implementation

In the original reference model, all OFDM symbols within a burst are used to calculate $P_{comp}$. This means that all symbols would have to be received before the first symbol could be compensated and demodulated. In reality this is not possible since it would require too much memory and the latency of the baseband processing would be too long. Therefore we decided to look at the trend of the 8 pilot subcarriers from three symbols - the present and the two previously received symbols.

The original model assumes that there is no timing drift within a burst, i.e. $C(4)$ is zero. In reality this is a coarse approximation but since the model has been changed to only consider three symbols, the approximation becomes more realistic since the timing drift over three symbols is much less than during a burst of several symbols. Thus the modification of the model is justified.

An overview of how the phase tracking is completed is presented in Figure 28.

![Figure 28 – Phase tracking tasks](image)

7.4.5.1 Pilot subcarrier phase rotation

The received pilot subcarrier phases are de-rotated and the phase rotation for each pilot subcarrier, denoted $X$, are determined according to

$$X = (\text{received pilots}) \cdot (\text{known pilots}) \cdot \text{conj} \{\text{channel estimate}\}$$

Eq. 7-18

In fact, each element in vector $X$ is a complex number and it is the angle of the complex number that represents the phase rotation. Since the phase rotation of a pilot subcarrier is obtained from a multiplication between complex numbers, also the amplitudes of the received pilots and the channel estimate have been taken into consideration. This means that a received pilot subcarrier with small amplitude counts less than one with large amplitude, but also that the reliability of the channel that the pilot subcarrier was transmitted on is of significance.

The transmitted pilot subcarriers are simply ±1 (BPSK) which simplifies the equation above to only one multiplication between the received pilots and the conjugated channel estimate. The result should then be negated if the transmitted pilot subcarrier was -1. The implementation is done as follows:

The CM, holding the channel estimate, is set up to point to the address the pilot subcarriers and the negation of the pilots is done by bringing the samples into the CMAC unit. A DM holding the
received OFDM symbol is setup to point its pilot subcarriers, i.e. point to addresses \( \text{mod}_{256}\{-88, -63, -38, -13, 13, 38, 63, 88\} \). \( X \) is obtained after a vector multiplication of length 16. Double read transfer can not be used because the pilot subcarriers are not located on adjacent frequency offset indices, i.e. memory addresses.

The content of \( X(i) \) should be interpreted as the phase rotation of pilot subcarrier with frequency offset index \( k \) for \( i \in \{1,2,3,...,8\} \) and \( k \in \{-88,-63,-38,-13,13,38,63,88\} \).

The vector of phase increments/decrements is calculated in 50 lines of assembly code and requires 78 clock cycles to complete.

### 7.4.5.2 Frequency dependent phase ramp of pilot subcarriers

A subcarrier with frequency offset index \( k \) experiences the same amount of frequency offset as the subcarrier with frequency offset index \(-k\). This is exploited and used to extract a phase ramp among the pilot subcarriers that only depends on frequency offset between the transmitter and the receiver, i.e. the phase ramp is independent of timing drift of the OFDM symbol. The phase ramp, denoted \( Z_T \), is achieved by adding the phase rotations for pilot subcarriers with frequency offset index \( \pm k \), according to

\[
Z_T = \sum X(1:1:4) \cdot X(8:(-1):5)
\]

**Eq. 7-19**

Vector \( X \) was a result of a vector instruction and is hence located in a DM. Half of the pilot subcarrier rotations must be moved to a different memory used for intermediate storage since a vector multiplication reads data from two separate memories. Moving data is done with the vector instruction \( \text{vmove} \). Then both memories are setup for a multiply-accumulate instruction, according to Eq. 7-19, after which the result, obtained in an accumulator, is stored to the CM.

The frequency dependent phase ramp is calculated in 40 lines of assembly code and requires 44 clock cycles to complete.

### 7.4.5.3 Time dependent phase ramp of pilot subcarriers

With the same discussion as for \( Z_T \), a time dependent phase ramp that is independent of a frequency offset can be achieved. This phase ramp is obtained by subtracting the phase rotation between pilot subcarrier with frequency offset index \( \pm k \) according to

\[
ZF = \sum X(1:1:4) \cdot \text{conj}\{X(8:(-1):5)\}
\]

**Eq. 7-20**

The implementation is done by enabling the conjugate flag and then performing the same operations as when calculating \( Z_T \), hence the same cycle cost is obtained.
7.4.5.4 Average phase rotation

The average phase rotation for each subcarrier, independent of the OFDM symbol number, is determined as

\[ C(1) = \text{angle}\left\{ \sum ZT \cdot e^{-j(\text{OFDM symbol number})C(2)} \right\} \]  \hspace{1cm} \text{Eq. 7-21} \]

The interpretation of Eq. 7-21 is as follows. The frequency dependent phase ramp is compensated with respect to the phase difference between two subsequent OFDM symbols with the OFDM symbol number taken into consideration. Then the angle of the compensated frequency dependent phase ramp is the resulting average phase rotation for each subcarrier.

The values of \( C(2) \), explained in section 7.4.5.5, and \( ZT \) are stored in the CM as described earlier and they are moved to two separate data memories. Multiplication between \( C(2) \) and the OFDM symbol number is performed as a repeated addition of \( C(2) \) in the ALU and the result is the exponent of the complex exponential function. The 9 MSB of the 16-bit register holding the angle are used as input for the look-up table (of size \( 2^9 \)) located in the CM. Then a multiplication between \( ZT \) and the complex exponential function value is made. The conjugate flag is enabled (before the multiplication) only if \( C(2) \) is positive, i.e. the exponent will be negative due to the minus sign in the equation since the OFDM symbol number is a positive integer. The compensated \( ZT \)-value is stored back to the CM in the same manner as described earlier where after a summation of the three most recent \( ZT \)-values is done. The angle of the resulting summation is straightforward calculated with the CORDIC subroutine. The result, \( C(1) \), is stored in the CM.

The average phase rotation is calculated in 80 lines of assembly code and requires 263 clock cycles to complete.

7.4.5.5 Phase difference of subcarriers with respect to time

By determining the trend of the frequency dependent phase ramp among the pilot subcarriers and then taking the angle, a phase difference of a subcarrier with respect to time, i.e. OFDM symbol number, denoted \( C(2) \), can be determined.

\[ C(2) = \text{angle}\left\{ \sum ZT(2 : \text{end}) \cdot \text{conj}(ZT(1 : (\text{end} - 1))) \right\} \]

\[ = \text{angle}\{ZT(i - 1) \cdot \text{conj}[ZT(i - 2)] + ZT(i) \cdot \text{conj}[ZT(i - 1)]\} \]  \hspace{1cm} \text{Eq. 7-22} \]

where \( i \) correspond to the index of the current received OFDM symbol.

To be able to determine \( C(2) \) at least two \( ZT \)-values are needed. The calculation includes three main operations, angle calculation, multiplication and summation. The angle calculation uses the previous explained CORDIC algorithm. The multiplication and summation could use the vector operation multiply-accumulate but the \( ZT \)-values are located in one memory while the vector instruction reads data from two separate memories. One solution to overcome the memory conflict would be to move two \( ZT \)-values into another memory, but using one multiplication (mul.1) and one accumulation (acc.1) is a more effective solution of the problem.
Now, for every new received OFDM symbol, we only need to perform one multiplication between the current \( ZT \)-value and the previously calculated \( ZT \)-value where after the result is saved. Since only the three last \( ZT \)-values are used, the results of the two multiplications are added together.

An example is used to explain the implementation. If the current index of the received OFDM symbol is 3, then we have previously (when \( i \) was 2) calculated and stored \( ZT(2) \cdot \text{conj}(ZT(1)) \). Now only \( ZT(3) \cdot \text{conj}(ZT(2)) \) is calculated and stored. Since a \text{mul.1} is used, the result of the multiplication is still in the accumulator of the CMAC. By using the \text{acc.1} instruction the previously stored result of \( ZT(2) \cdot \text{conj}(ZT(1)) \) are added to the accumulator and the angle of result of the accumulation is obtained with the CORDIC algorithm. The resulting angle is then stored to the CM as \( C(2) \).

The average phase difference is calculated in 50 lines of assembly code and requires 237 clock cycles to complete.

**7.4.5.6 Phase difference between two subsequent subcarriers**

All the time dependent phase ramps can be summed into a vector representing a time dependent phase ramp for the entire burst. If the angle of the ramp is determined and compensation is made with respect to the location of the pilot subcarriers within the OFDM symbol, a phase difference between two subsequent subcarriers, denoted \( C(3) \), can be found.

\[
C(3) = \frac{\text{angle} \left( \sum \text{ZF} \right)}{\sum |\text{Pilot sub carrier number}|} \quad \text{Eq. 7-23}
\]

The sum of the three \( ZF \)-values is done with the vector operation \text{acc.3} and then the CORDIC subroutine is used to determine the angle. One over the denominator and the minus sign in the equation above is saved as a constant in the CM to avoid a division. This means that only the multiplication between the angle and the constant in CM needs to be calculated.

The phase difference between two subsequent subcarriers is calculated in 31 lines of assembly code and requires 223 clock cycles to complete.
7.4.5.7 Compensation of data subcarriers

The total phase rotation, $P_{\text{comp}}$, is used to compensate the received data OFDM symbols as in Eq. 7-24 which results in a de-rotation of the data subcarriers.

\[
(\text{Compensated data}) = (\text{received data}) \cdot e^{-jP_{\text{comp}}}
\]  

Eq. 7-24

To reduce the number of memory accesses, and thereby the cycle cost, we decided to put the $P_{\text{comp}}$ calculation and the compensation together as one operation. A look-up table is used for the compensation in the same manner as when calculating $C(1)$. This means that $P_{\text{comp}}$ is just an address for the look-up table holding the complex exponential function. First $C(1) + C(2) \cdot (\text{OFDM symbol number})$ is calculated because this is constant for all subcarriers during one OFDM symbol. To calculate $P_{\text{comp}}$ for subcarrier with frequency offset index 1, $C(3)$ is simply added, and for subcarrier with frequency offset index 2 another $C(3)$, etc. For the negative subcarriers the procedure is repeated but with subtraction of $C(3)$ instead of addition. The memory read address for the look-up table in CM is determined by the 9 MSB of the register holding the resulting angle. Then a mul.1 is performed between the data and the value in CM pointed out by $P_{\text{comp}}$. The minus sign in the exponent is implemented by setting the conjugate flag before the multiplication. The negative subcarriers are stored in increasing order with the most negative frequency offset index first. Since $C(3)$ is subtracted from the former result we start with compensation of subcarrier with frequency offset index -1, which has the highest address, and decrease the address for every subcarrier. The processor does not support memory read/write address decrement so the read and write address must be updated for every negative subcarrier which takes 4 clock cycles extra. The operations needed for every subcarrier are implemented as subroutines, one for positive and one for negative subcarriers. The subroutines will be called by loops that are 9 respectively 13 cycles long after scheduling. There are 192 subcarriers that need to be compensated so the loops will be looped 96 times each which gives 96·(9+13) = 2112 clock cycles. The data to be sent to the demodulator should not include the eight pilot subcarriers hence there is no compensation for them.

The compensation of all data subcarriers within an OFDM symbol is done in 141 lines of assembly code and requires 2215 clock cycles to complete.
7.4.5.8 Resource allocation

One DM holds the 256 frequency domain samples of the received OFDM data symbol. The same memory is also used to store intermediate results from vector instructions, and in total 260 addresses is allocated within this memory.

A second DM is used as a workspace to store intermediate results of vector instructions. Only four addresses are used in this memory.

The CM is used to store the different phases that are needed during the de-rotation of the subcarriers and some intermediate result. In total the required memory allocation of the CM is determined to 22 addresses and the content is presented in Table 8.

<table>
<thead>
<tr>
<th>Number of used memory address</th>
<th>Memory content</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>Extracted pilot subcarriers from channel estimate</td>
</tr>
<tr>
<td>3</td>
<td>Three last $ZT$-values</td>
</tr>
<tr>
<td>3</td>
<td>Three last $ZF$-values</td>
</tr>
<tr>
<td>3</td>
<td>$C(1)\cdot C(2)\cdot C(3)$</td>
</tr>
<tr>
<td>1</td>
<td>$ZT(i-1)\cdot \text{conj}(ZT(i-2))$</td>
</tr>
<tr>
<td>3</td>
<td>$ZT\cdot e^{j\text{symbol number}}\cdot C(2)$</td>
</tr>
<tr>
<td>1</td>
<td>Constant used for calculation of $C(3)$</td>
</tr>
</tbody>
</table>

Table 8 – CM content: phase tracking and compensation
7.4.5.9 Performance

Table 9 shows the performance of the implemented phase tracking and compensation algorithm.

<table>
<thead>
<tr>
<th>Task</th>
<th>Lines of assembly code</th>
<th>Clock cycles</th>
<th>Percent of total clock cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pilot subcarrier phase rotation - X</td>
<td>50</td>
<td>78</td>
<td>2.5</td>
</tr>
<tr>
<td>Frequency dependent phase ramp - ZT</td>
<td>40</td>
<td>44</td>
<td>1.4</td>
</tr>
<tr>
<td>Time dependent phase ramp - ZF</td>
<td>40</td>
<td>44</td>
<td>1.4</td>
</tr>
<tr>
<td>Average phase rotation – C(1)</td>
<td>80</td>
<td>263</td>
<td>8.5</td>
</tr>
<tr>
<td>Phase difference of subcarriers with respect to time – C(2)</td>
<td>50</td>
<td>237</td>
<td>7.6</td>
</tr>
<tr>
<td>Phase difference between two subsequent subcarriers – C(3)</td>
<td>31</td>
<td>223</td>
<td>7.2</td>
</tr>
<tr>
<td>Compensation - P_{comp}</td>
<td>141</td>
<td>2215</td>
<td>71.4</td>
</tr>
<tr>
<td>Complete phase tracking with compensation</td>
<td>432</td>
<td>3104</td>
<td>100</td>
</tr>
</tbody>
</table>

Table 9 – Performance of phase tracking and compensation

The approximate signal to quantization ratio, SQR, is calculated as

\[
SQR_{Phase\;tracking} = 20 \cdot \log_{10} \left( \frac{\sum \left| \text{Phase\;tracking}_{\text{Matlab}} \right|}{\sum \left| \text{Phase\;tracking}_{\text{Matlab}} - \text{Phase\;tracking}_{\text{Processor}} \right|} \right)
\]

An SQR of 42.2 dB after compensation of the received data was achieved and corresponds to having 7 correct bits representation of the subcarrier data samples. It should be pointed out that this is the least number of correct bits and it does not represent an average, the average number of correct bits would be more than 7. The following stage for the compensated data is through the demodulator which outputs soft bits that are quantized to a fixed number of bits for the forward error correction (FEC) block. In general, the number of bits used to represent each soft bit in the FEC block is six or seven; hence the result of the phase tracking and compensation implementation is acceptable.
7.5 OFDM symbol demodulation

At the transmitter modulation is the process of mapping digital information to analog form so it can be transmitted over the channel. Consequently the receiver performs the inverse process, demodulation, which reproduces digital information from the received OFDM symbols. [13].

Modulation is performed by varying the amplitude and/or phase of the transmitted RF carrier signal in accordance with the digital information that is to be transmitted. The transmitted signal voltage to the antenna during any OFDM symbol is [6]

\[
s(t) = \text{Re}\left\{ e^{j2\pi f_c t} \cdot \sum_{k=-N_{\text{used}}/2}^{N_{\text{used}}/2} c_k \cdot e^{j2\pi k\Delta f(t-T_g)} \right\}
\]

which results in both in-phase (I) and quadrature (Q) carrier waves that depends on \(c_k\) where

- \(t\) is the time elapsed since the beginning of the OFDM symbol.
- \(f_c\) is the signal carrier.
- \(k\) is the subcarrier frequency offset index for data to be transmitted on.
- \(N_{\text{used}}\) is the number of used subcarriers, i.e. 200.
- \(\Delta f = F_s/N_{\text{FFT}}\) is the subcarrier spacing.
- \(T_g\) is the time of the cyclic prefix.
- \(c_k\) is a complex number that depends on the digital information that is to be transmitted.

Modulation of the digital information is made by delimiting the binary data into groups of \(i\) bits representing a symbol\(^3\). Hence each symbol is one of \(2^i\) possible points and the total number of points is referred to as a constellation. The IEEE 802.16d standard supports BPSK, QPSK, 16-QAM and 64-QAM constellations where the sizes of the delimited groups are 1, 2, 4 and 6 bits respectively. The four constellations are presented below where the constellation points have assigned bit patterns, with \(b_0\) denoting the LSB [6]. The constellations are presented in the I/Q plane, where the values of I and Q are interpreted as the real and imaginary parts of \(c_k\).

---

\(^3\) Not to be confused with an OFDM symbol.
The received symbols will most likely not be located at the exact origin in the constellation due to added noise during transmission of the signal. The receiver must be able to decide what symbol that was actually sent to be able to recover the conveyed digital information.
7.5.1 Demodulation example

The following figures illustrate the task of demodulation.

Assume that the transmitted digital information is \(b_3b_2b_1b_0 = 0000\) resulting in a constellation point \([I,Q] = [1,1]\) and let the “X” in Figure 33 mark the received symbols I and Q values.

If the received in-phase signal is larger than zero the transmitted symbol can be assumed to be located in the left half plane \((I > 0)\) where all symbols has a 0 as its leftmost bit. Hence the decision is \(b_3 = 0\). This is illustrated in Figure 34.

In the same manner, a received quadrature signal larger than zero sets the decision of \(b_1 = 0\) since all symbols in the upper half plane \((Q > 0)\) has a 0 as the third bit from the left. This is illustrated in Figure 35. Now the received symbol is determined to be in the first quadrant of the constellation.
Furthermore, the decision boundaries at \( I = \pm 2 \) are used to determine \( b_2 \) since, if the distance between \( I = 2 \) and the received symbol \( I \) value and is smaller than 2 then \( b_2 = 0 \) since all symbols with \(-2 < I < 2\) have a 0 as the second bit from the left. This is illustrated in Figure 36.

At last, the decision boundaries at \( Q = \pm 2 \) are used to determine \( b_0 \) since all symbols with \(-2 < Q < 2\) have a 0 as its rightmost bit if the distance between \( Q = 2 \) and the received symbol \( Q \) value is smaller than 2. This is illustrated in Figure 37.

The presented example of demodulation makes a definite determination of whether a 0 or 1 was transmitted. This is said to be a hard decision and the output from the demodulator is 0000\(_2\) with no extra information about the reliability of the decision.

### 7.5.2 Ramesh algorithm

The implementation of the demodulator is based on Ramesh algorithm for soft bit generation. This type of demodulator includes additional information about the decision reliability by
producing an output were the sign indicates a 0 or 1 and the absolute value is the distance to the
decision boundary. [15].
In the previous example, not much additional noise in the received signal would place the “X” in
the lower half plane and then b1 would flip to 1. Also, if “X” is moved to the region where I > 2
then b2 will flip to 1. Since soft bits are used, the short distance from “X” the decision boundaries
can be taken for considered by the error correction algorithms and their performance will be
significantly improved.

7.5.3 Implementation
Depending on the number of coded bits per subcarrier, i.e. number of bits of a constellation point,
the demodulation process is different but will include the same type of operations. The algorithm
for demodulation of a 16-QAM constellation symbol is presented to point out the operations done
in LeoCore.

At first the channel estimate is used to compensate the received data with respect to its subcarrier
frequency offset index, and to adjust the decision boundaries.

\[ X = \text{conj}(\text{channel estimate}) \cdot (\text{received data}) \]  

\[ K = (\text{decision boundary}) \cdot |\text{channel estimate}|^2 \]  

Then the soft bits are generated according to

\[ b_3 = \text{real}(X) \]  
\[ b_1 = \text{imag}(X) \]  
\[ b_2 = K - |b_3| \]  
\[ b_0 = K - |b_1| \]  

The channel estimate was previously stored in the CM, and the received data is located in a DM.
During the preceding phase compensation of the received OFDM symbol, data has been
rearranged in such way that the 192 data subcarriers are stored on adjacent memory addresses
which enables for efficient usage of vector instructions.

The calculation of X is straightforward implemented with mul.192 while the conjugate flag of the
CMAC is enabled. The result is a complex vector with real and imaginary parts interpreted as b3
and b1, respectively.

Calculation of the square of the absolute value of the channel estimate is done using sqrabs.192.
The decision boundary is a real constant stored in the CM. The equations for the soft bits \( b_2 \) and \( b_0 \) can be rewritten as:

\[
\begin{align*}
b_2 &= \text{real}\{K \cdot (1 + j) - (|\text{real}\{X\}| + j \cdot |\text{imag}\{X\}|)\} = \text{real}\{(K - |\text{real}\{X\}|) + j \cdot (K - |\text{imag}\{X\}|)\} \\
b_0 &= \text{imag}\{K \cdot (1 + j) - (|\text{real}\{X\}| + j \cdot |\text{imag}\{X\}|)\} = \text{imag}\{(K - |\text{real}\{X\}|) + j \cdot (K - |\text{imag}\{X\}|)\}
\end{align*}
\]

Eq. 7-28

Instead of storing the original real constant we store the constant as a complex number which results in a complex vector \( K \) with identical real and imaginary parts. The absolute values of the real and imaginary parts of \( X \) are calculated, which corresponds to calculating \( |b_3| \) and \( |b_1| \), after which a new vector \( Y \) is obtained.

\[
Y = |\text{real}(X)| + j \cdot |\text{imag}(X)|
\]

Eq. 7-29

A complex vector subtraction is done between \( K \) and \( Y \), resulting in a new complex vector from which we interpret \( b_2 \) and \( b_0 \) as the real and imaginary parts respectively. With this implementation we avoid moving data between memories without performing an actual operation.

7.5.3.1 Resource allocation

In total, two DMs and the CM must be used, since vector instructions reads data from two separate memories and writes the result to a third memory. The memory allocation is made as follows.

One DM holds the present data OFDM symbol of 192 samples. The same memory is used as a workspace and intermediate results are placed here. In total only 192 memory addresses are used for this memory.

Another DM contains the
- \( X \) vector,
- \( |\text{Channel estimate}|^2 \) vector,
- \( Y \) vector (= \( |\text{real}(X)| + j*|\text{imag}(X)| \)),
- and a vector similar to \( Y \).

All vector sizes are 192 samples and, together with the constants, 768 memory addresses are used for this memory.

The CM holds the channel estimate, the three decision boundary constants for 16- and 64-QAM demodulation and the results of two vector subtractions. Since double read transfer is used by the vector instructions, the constants are stored on two memory addresses each. All vectors are of size 192 samples. The required CM allocation is determined to 390 addresses.
7.5.3.2 Performance

Table 10 shows the performance of the implemented demodulation algorithm which is completed in 150 assembly lines for the largest constellation.

<table>
<thead>
<tr>
<th>Modulation</th>
<th>Total number of soft bits to determine per OFDM symbol</th>
<th>Clock cycles</th>
<th>Approximate SQR [dB]</th>
</tr>
</thead>
<tbody>
<tr>
<td>BPSK</td>
<td>192</td>
<td>110</td>
<td>59</td>
</tr>
<tr>
<td>QPSK</td>
<td>384</td>
<td>110</td>
<td>59</td>
</tr>
<tr>
<td>16-QAM</td>
<td>768</td>
<td>1782</td>
<td>55</td>
</tr>
<tr>
<td>64-QAM</td>
<td>1152</td>
<td>3356</td>
<td>53</td>
</tr>
</tbody>
</table>

Table 10 – Performance of the demodulation implementation

It can be seen that to demodulate a BPSK or QPSK constellation point requires the same amount of time even though the number of soft bits to be produced differs with a factor 2. This is due to the fact that calculation of the vector X is done in both cases even though only the real part is needed for BPSK.

For both QAM-constellations a dramatic increase of clock cycles is obtained and this is due to the calculation of \(|b|\). The instruction set of the processor does not include any instruction for calculation of the absolute value of a vector. The operation must therefore be done as follows:

Loop 96 times:
- Load two accumulator with two complex samples from memory holding vector X. (cycle cost: 2)
- Move real and imaginary parts of the two samples from the accumulators to four general registers. (cycle cost: 4)
- Calculate the absolute values of the four general registers. (cycle cost: 4)
- Move the four general registers back to the accumulators into two complex samples. (cycle cost: 4)
- Store the two complex samples to memory holding vector Y. (cycle cost: 1)

The total cost of calculating the absolute values is 1441 clock cycles and this can be compared with executing a vector instruction of size 192, which is done in only 101 clock cycles.

The actual implementation of the absolute operation exploits that a load instruction requires two clock cycles to complete, i.e. the load instruction is followed by a no-operation instruction, which enables for instruction scheduling. This means that each loop reads twice as many complex samples and the instructions are re-arranged in such way that the load instruction always is followed by an arithmetic operation. With this scheduling, the cycle cost is reduced to 1346 clock cycles at the cost of double amount of used general registers and accumulators. This is not an issue since there are several unused general register. Still, this is a too high cycle cost for the absolute operation.
The implication of this is that a new vector instruction is needed in LeoCore’s instruction set for a more efficient 16- and 64-QAM demodulation.

At last, demodulation of 64-QAM includes another round of calculating the absolute value of a vector which explains the even higher cycle cost compared to 16-QAM demodulation.

However, despite the high cycle cost, the resulting SQR is at lowest 53 dB which is corresponds to 8 correct bits representation of the soft bits. The SQR is calculated as

\[
SQR_{\text{Demodulation}} = 20 \cdot \log_{10} \left( \sum \left| \frac{\text{Soft bits}_{\text{Matlab}}}{\sum |\text{Soft bits}_{\text{Matlab}} - \text{Soft bits}_{\text{Processor}}|} \right| \right)
\]
8 Receiver integration

The different tasks that have been implemented constitute the fundamentals of baseband processing in an OFDM system. A simulation of how the different tasks would function if they were to be integrated into a complete receiver is presented in this section.

An overview of the simulation settings is shown in Figure 38. The Matlab model generates a burst of OFDM symbols that is passed through a channel. Output data from the channel is saved and passed on to CDS as input files for the DMs of the processor. Baseband processing is performed in the processor and the result, i.e. soft bits, is written to file and passed back to Matlab for comparison with Matlab’s result of the receiver output.

![Figure 38 – Simulation](image-url)
8.1 Transmitter
The transmitter is setup to randomly generate QPSK or 16-QAM constellation-mapped data into a time-domain signal.

8.2 Channel
The signal passes through a channel that adds
- White Gaussian noise with a given SNR (specified to be 20 dB).
- A time delayed version of the signal (5 samples delay).
- A phase ramp caused by the carrier frequency offset which is specified to be 625 Hz.
- A timing drift error caused by the carrier frequency offset.

8.3 Receiver
The receiver architecture is presented in Figure 39.

![Figure 39 – Receiver architecture](image)

Each block has been implemented separately as an independent project in CDS. Packet detection is not implemented since it is assumed to be part of the DFE, so the coarse timing instant for the preamble is assumed to be known. During the simulation of the integrated receiver, the memory content after a substage is saved and used as input to the subsequent substage. Further resource allocation with respect to data memory usage and the usage of the general registers would be necessary. This is not done in this thesis because the main concern here is to evaluate baseband processing algorithms on the processor architecture.

However, the baseband processing of a complete receiver could be implemented as follows. The DFE would generate an interrupt when the 4x64 preamble is detected. The following 288 data samples, representing $P_{EVEN}$, would be transferred to a DM. When $P_{EVEN}$ is received, the subsequent data OFDM symbol samples coming from the DFE would be re-directed to a different DM. Meanwhile, the received samples of $P_{EVEN}$ would go through the FFT and channel estimation block.

For every received data OFDM symbol the procedure is repeated. While the samples corresponding to symbol $l+1$ are being received to a separate DM, baseband processing of
symbol $l$ is performed using two different DMs. The baseband processing of a data OFDM symbol consists of FFT, phase tracking and compensation, and demodulation. These are the tasks that must be completed within the time it takes to receive an OFDM symbol.

During the baseband processing, care must be taken so that neither of the receiver blocks writes to the DM currently receiving the next data OFDM symbol. Each receiver block uses at most two DMs and one DM is reserved for reception of the next OFDM symbol. Since the processor originally has four DMs, one is left unused. The ulterior motive for this is that the processor can make use of symbol pipelining if accelerators for de-interleaving, FEC-decoding and de-randomization are added. Then the processor core can focus on the tasks that are implemented during this thesis and hand over to a chain of accelerators where after it can proceed with the next OFDM symbol.

8.3.1 Simulation results
The comparison block in Figure 38 compares the reference model with the processor, and plots data between the different substages of the processor.

Figure 40 and Figure 41 show the received signal QPSK and 16-QAM constellation after FFT in LeoCore. The constellation points can not be discerned from each other and the transmitted data would not be recovered during the process of demodulation.

![QPSK constellation symbols before phase tracking and compensation](image1)

![16-QAM constellation symbols before phase tracking and compensation](image2)
Figure 42 and Figure 44 show the QPSK and 16-QAM constellation symbols after de-rotation of the subcarriers in LeoCore. As seen, the constellations are warped but yet legible. This emphasizes the importance of synchronization algorithms in OFDM systems. If the synchronization algorithms are too inaccurate this will lead to that the constellation symbols are too spread out. The results from LeoCore can be compared with Matlab’s results in Figure 43 and Figure 45.

Figure 42 – QPSK constellation symbols after phase tracking and compensation in LeoCore

Figure 43 – QPSK constellation symbols after phase tracking and compensation in Matlab

Figure 44 – 16-QAM constellation symbols after phase tracking and compensation in LeoCore

Figure 45 – 16-QAM constellation symbols after phase tracking and compensation in Matlab
Figure 46 shows the equalized QPSK constellation symbols from LeoCore, i.e. the symbols are compensated with respect to the channel estimate as done by the demodulator. The result after equalization is a fully legible QPSK constellation and the constellation symbols are far away from their decision boundaries. The result from LeoCore can be compared with Matlab’s result in Figure 47.

![Figure 46 – Equalized QPSK constellation symbols in LeoCore](image1)

![Figure 47 – Equalized QPSK constellation symbols in Matlab](image2)

Figure 48 shows the equalized 16-QAM constellation symbols from LeoCore. In comparison with QPSK, some 16-QAM constellation points are fairly more spread out and located closer to their decision boundaries. If some constellation symbols cross their decision boundaries, they will be interpreted to origin from a different constellation point during the demodulation process, yielding incorrect soft bits. The result from LeoCore can be compared with Matlab’s result in Figure 49.

![Figure 48 – Equalized 16-QAM constellation symbols in LeoCore](image3)

![Figure 49 – Equalized 16-QAM constellation symbols in Matlab](image4)
Figure 50 shows a histogram of the soft bits from the QPSK constellation produced by LeoCore, i.e. the output from the demodulator. The horizontal axis represents the likely distribution and the vertical axis represents quantity. The result from LeoCore can be compared with Matlab’s result in Figure 51. As seen, the apexes are well separated and the FEC decoding block should be able to determine the correct transmitted information from the soft bits.

![Figure 50 – Histogram of soft bits from QPSK constellation symbols produced by LeoCore](image1)

![Figure 51 – Histogram of soft bits from QPSK constellation symbols produced by Matlab](image2)

Figure 52 shows the histogram of the soft bits from the 16-QAM constellation produced by LeoCore. The apexes are also here well separated and thus retrieval of the transmitted information should be possible, even if the apexes are closer together compared with the histogram for QPSK soft bits. The result from LeoCore can be compared with Matlab’s result in Figure 53.

![Figure 52 – Histogram of soft bits from 16-QAM constellation produced by LeoCore](image3)

![Figure 53 – Histogram of soft bits from 16-QAM constellation produced by Matlab](image4)
The simulation results from LeoCore are presented in Table 11 and Table 12.

<table>
<thead>
<tr>
<th>Task</th>
<th>Approximate SQR [dB]</th>
<th>Number of clock cycles</th>
<th>Lines of assembly code</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFT</td>
<td>46</td>
<td>1117</td>
<td>109</td>
</tr>
<tr>
<td>Channel estimation</td>
<td>54</td>
<td>4975</td>
<td>444</td>
</tr>
<tr>
<td>Phase tracking and compensation</td>
<td>40</td>
<td>3104</td>
<td>432</td>
</tr>
<tr>
<td>Demodulation</td>
<td>39</td>
<td>110</td>
<td>16</td>
</tr>
</tbody>
</table>

Table 11 – Simulation results: QPSK signal

<table>
<thead>
<tr>
<th>Task</th>
<th>Approximate SQR [dB]</th>
<th>Number of clock cycles</th>
<th>Lines of assembly code</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFT</td>
<td>46</td>
<td>1117</td>
<td>109</td>
</tr>
<tr>
<td>Channel estimation</td>
<td>54</td>
<td>4975</td>
<td>444</td>
</tr>
<tr>
<td>Phase tracking and compensation</td>
<td>35</td>
<td>3104</td>
<td>432</td>
</tr>
<tr>
<td>Demodulation</td>
<td>33</td>
<td>1782</td>
<td>80</td>
</tr>
</tbody>
</table>

Table 12 – Simulation results: 16-QAM signal

It can be seen that the performance is somewhat better for the QPSK signal. An SQR of 39 dB of the soft bits corresponds to having slightly more than 6 bits correct representation compared to Matlab’s floating point representation of the soft bits. For the 16-QAM signal, the soft bits are produced with an SQR of 33 dB which corresponds to having 5 correct bits.
8.3.2 Total resource allocation

Table 13 shows the memory usage of LeoCore and in total 11.3 kilobytes have been used.

<table>
<thead>
<tr>
<th>Memory</th>
<th>Size [bytes]</th>
</tr>
</thead>
<tbody>
<tr>
<td>PM</td>
<td>2270</td>
</tr>
<tr>
<td>CM</td>
<td>3540</td>
</tr>
<tr>
<td>3 DMs</td>
<td>5376</td>
</tr>
<tr>
<td>IM</td>
<td>96</td>
</tr>
</tbody>
</table>

Table 13 – LeoCore memory usage

The size of the PM is determined by the total number of assembly lines: 1000 assembly lines equal 500 bytes.

All pre-calculated values such as filter taps, twiddle factors, look-up table, various constants etc. are stored in the CM. A coarse overview of the content is presented in Table 14, and in total 885 memory addresses are used.

<table>
<thead>
<tr>
<th>Number of addresses</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td>FFT twiddle factors</td>
</tr>
<tr>
<td>6</td>
<td>CORDIC arctangent values</td>
</tr>
<tr>
<td>256</td>
<td>Look-up table for complex exponential function</td>
</tr>
<tr>
<td>211</td>
<td>Channel estimation</td>
</tr>
<tr>
<td>22</td>
<td>Phase tracking</td>
</tr>
<tr>
<td>6</td>
<td>Demodulation</td>
</tr>
<tr>
<td>384</td>
<td>Temporary workspace</td>
</tr>
</tbody>
</table>

Table 14 – Total CM content

The IM holds the arctangent values that are used during angle calculations.

The size of the DM storage is the maximum allocated memory space by any of the receiver tasks that have been implemented. An OFDM symbol consists of 288 time-domain samples and the maximum allocated memory space is 768 (during the demodulation process), hence (768+288+288)/32/8 is the memory size in bytes.
8.3.3 Clock frequency

The different tasks of the receiver should be completed within the time it takes to receive an OFDM symbol. The different tasks are completed in $T_{LeoCore}$ seconds.

$$T_{LeoCore} = \frac{\sum \text{clock cycles}}{\text{clock frequency}}$$  \hspace{1cm} \text{Eq. 8-1}

The number of clock cycles (cc) is determined by

$$\max\{cc_{FFT} + cc_{Channel est} \quad ; \quad cc_{FFT} + cc_{Phase tracking + comp} + cc_{Demodulation}\} =$$

$$\max\{1117 + 4975 \quad ; \quad 1117 + 3104 + 3356\} = 7577 \text{ clock cycles}$$ \hspace{1cm} \text{Eq. 8-2}

The time-domain OFDM symbol period, according to Table 1, is

$$T_s = 72 \mu s$$ \hspace{1cm} \text{Eq. 8-3}

Hence, the required clock frequency for LeoCore is determined to

$$f_{CLK} \geq \frac{7577}{72 \cdot 10^{-6}} \approx 105 \text{ MHz}$$ \hspace{1cm} \text{Eq. 8-4}

This is the minimum clock frequency required if only the above tasks are to be completed while receiving the next OFDM symbol. If more tasks would be implemented in LeoCore, such as de-interleaving, FEC decoding etc, the clock frequency must be increased. To avoid this, accelerators should be considered for such tasks. However, LeoCore can be used with clock frequencies over 200 MHz which means that possible additional workload can be as much as the tasks already implemented.
9 Conclusions

9.1 Evaluation of LeoCore

LeoCore is a baseband processor technology and it is when it comes to baseband specific calculations, like FFT and vector energy, etc, it highlights its strengths. With one line of assembly code, and in most cases a few lines for network and memory setup, complex operations involving several hundreds of calculations can be initialized. When it comes to operations that handles single or non subsequent data it becomes obvious that the processor is not optimized for such tasks. The assembly program becomes relatively long for an instruction that operates on single data with respect to what it actually accomplishes. When data to be processed is stored on non consecutive memory addresses the effectiveness of the vector instructions are reduced. A vector instruction of twice the length must be used since the CMAC reads two consecutive addresses during each transaction. It also means that twice as many results are calculated and must be handled.

; Network connections
nwc dm0, port1
nwc dm1, port2

; Memory setup
acl dm0, #mem_rstep|2
acl dm0, #mem_read|0
acl dm1, #mem_rstep|2
acl dm1, #mem_read|0
mac.256 port2, port1

Figure 54 – Example of mac.256

; Network connections
nwc dm0, port1
nwc port2, dm1
nwc cm, port0

; Memory setup
acl dm0, #mem_rstep|4
acl dm0, #mem_read|0
acl dm1, #mem_wstep|1
acl dm1, #mem_write|0
acl cm, #mem_rstep|1
acl cm, #mem_read|0
mul.3 port0, port1

Figure 55 – Example of mul.3

Figure 54 shows an example of the simplicity to execute a vector instruction that performs 512 operations, i.e. 256 multiplications and 256 additions, on data stored in subsequent order.

Figure 55 is an example of how to multiply data on address 0 and 4 in dm0 with data on address 0 and 1 in cm, respectively. The actual operations performed by the mul.3 instruction are:

1. Multiplication between dm0 address 0 and cm address 0, result stored to dm1 address 0.
   Multiplication between dm0 address 1 and cm address 1, result stored to dm1 address 1.

2. Multiplication between dm0 address 4 and cm address 1, result stored to dm1 address 1.

The multiplication in step 2 overwrites the second result calculated in step 1. Another way of performing the same calculation would to use two mul.1 instructions, but that will need some extra clock cycles due to instruction pipelining delay slots.
When all data to be processed by a vector instruction is located within the same memory, data must be moved. For example, if a data memory holds N (even integer) samples and sample 1 should be multiplied with sample N, sample 2 with sample N-1 and so on; a vector move of N/2 samples must be used before the actual multiplication can be executed. This is illustrated in Table 15 and Table 16, with N=8.

<table>
<thead>
<tr>
<th>Address</th>
<th>Content</th>
<th>Address</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Sample 1</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>Sample 2</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>Sample 3</td>
<td>2</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>Sample 4</td>
<td>3</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>Sample 5</td>
<td>4</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>Sample 6</td>
<td>5</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>Sample 7</td>
<td>6</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>Sample 8</td>
<td>7</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 15 – Initial data memory content

<table>
<thead>
<tr>
<th>Address</th>
<th>Content</th>
<th>Address</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Sample 1</td>
<td>0</td>
<td>Sample 5</td>
</tr>
<tr>
<td>1</td>
<td>Sample 2</td>
<td>1</td>
<td>Sample 6</td>
</tr>
<tr>
<td>2</td>
<td>Sample 3</td>
<td>2</td>
<td>Sample 7</td>
</tr>
<tr>
<td>3</td>
<td>Sample 4</td>
<td>3</td>
<td>Sample 8</td>
</tr>
<tr>
<td>4</td>
<td>Sample 5</td>
<td>4</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>Sample 6</td>
<td>5</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>Sample 7</td>
<td>6</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>Sample 8</td>
<td>7</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 16 – Data memory content after moving data

A vector instruction reads data from two memories and writes the result (when the result is not a scalar) to a third memory. If the input operands are stored in two DMs, the result must be written to CM since it is always connected to a port of the core. This is, in some cases, a restriction and sometimes it would be better if all three ports of the core were identical to enable for connection of three DMs to the core (instead of two DMs and the CM).

9.1.1 Pros
- The instruction set includes efficient vector instructions and enables for parallel computing.
- The assembly coding becomes simple and straightforward when the algorithms include operations covered by LeoCore’s instruction set.
- Vector instructions can be efficiently used when data is stored on consecutive memory addresses.

9.1.2 Cons
- Repeated processing of single data gives rise to large control overhead.
- Vector instructions of double length must be used when data is not stored on consecutive memory addresses.
- Vector instructions perform operations on data from two separate memories and moving data without performing an operation cannot always be avoided.

9.1.3 Efficient usage
To fully utilize the capability of LeoCore, the algorithms to be implemented should be using operations covered by the instruction set. This requires comprehension of LeoCore’s architecture when developing the algorithms. The assembly programmer should strive to arrange storage of data to be processed by a vector instruction on consecutive memory addresses. For example, if only channel frequency response samples for all even frequency offset indices would have been
required, the received samples of $P_{EVEN}$ could have been stored in an order so that samples corresponding to even indices were located on subsequent memory addresses instead of every other.

9.2 Final conclusions

The implemented parts have been successfully implemented and the required clock frequency was determined to 105 MHz.

9.3 Future work

There are a number of things that can be done to further improve the evaluation of LeoCore. The entire reference model of the receiver could be implemented to find out the needs of accelerators.

The SQRs calculated throughout the thesis relates the result from the reference model with the result from the processor. That means that the comparison is made with a result that is not achievable with finite word length. To find out if the calculated SQRs are adequate, quantization in the reference model could be introduced and the appropriate SQRs for the signals could be found.

Throughout the implementation safe scaling has been used. To achieve a higher SQR a different type of scaling which allows overflow with a certain probability could be introduced.

The digital front end is not supported by the simulator which results in that we have assumed an arbitrary small error from the packet detector. If this were to be simulated, the influence of the uncertainty in the packet detection could be evaluated in a more realistic sense.

As mentioned in chapter 9.1, the algorithms could be adjusted to be suitable for the LeoCore instruction set. This would better make use of the strengths within LeoCore’s technology.
References


Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extra-ordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/