Implementation of a LTE inspired Transceiver

On a USRP Platform

Group 12gr1056, Aalborg University
May 2012

AALBORG UNIVERSITY

The Faculties of Engineering, Science and Medicine
Department of Electronic Systems
Frederik Bajers Vej 7
Phone: +45 99 40 86 00
http://es.aau.dk

Title:
Implementation of a LTE inspired Transceiver on a USRP Platform
Theme:
Advanced Software Radio Implementation
Project period:
September 2, 2011 - May 31, 2012

Project group:
12gr1056

Group members:
Jakob L. Buthler
Michael Buhl

Supervisors:
Gilberto Berardinelli
Yannick Le Moullec

Number of copies: 5
Number of pages: 75
Appended documents:
4 Appendices, 1 CD-ROM
Total number of pages: 135
Finished: May 31th 2012

Abstract:

Long Term Evolution (LTE) is a upcoming telecommunication standard with throughput of up to 300 Mbps. Due to this it has been chosen as reference point, for analysing the possibilities of hardware accelerating the functionalities of the physical layer in a high performance communication system.
The analysed functionalities are the; CRC, Turbo Encoding, OFDM together with possible channel equalizer options such as ZF, MMSE and Turbo equalization.
Regarding implementation it is concluded that the USRP2 should be used as platform since this offers a variety of front-ends and an already existing FPGA image which handles the initial RF data processing such as down-sampling and filtering.
By analysing the functionalities performance on a CPU and FPGA, it is found that, especially, OFDM and the Turbo decoder will gain from hardware acceleration.
The initial focus of hardware implementation is OFDM since this can be done with transparent modules, which can easily be reused by others. Furthermore, the basic operation of OFDM, the FFT, is exhaustively researched and it does not require further communication between Host-PC and USRP than the already existing. To interface the block processing of the FFT and the continuous datastream of the FPGA, a pipeline module is designed which gathers symbols into packets.

Preface

This project is written in connection with the E9-10 semester in the master degree of electronics, with area of specialization in Software Defined Radio. The report is written by group 12gr1056 to document the effort put into the project through its 9 months duration (1. September to 31. May).

The project description was written by Andrea Fabio Cattoni in correspondence with the group members of 12gr1056.

We would like to thank our supervisors; Yannick Le Moullec and Gilberto Berardinelli for their constructive guidance in the different processes of the project and their continuously patience when receiving "almost finished" worksheets.

We would also like to thank our families and friends for their patience and constant encouragement, especially through the last phases of the process.

Michael Buhl

Jakob Lindbjerg Buthler

Reading Guide

It has been chosen to put some of the important analysis work, such as the implementational process of the hardware in the Appendix. This has been done to establish a flow in the report, such that it refers to conclusions, deductions and deducions made in the enclosed appendixes, which are meant to support the main report.

References to other works are made in the form [pagenumber, reference number] where the reference number refers to the given reference in the Bibliography Appendix. In the report, references are given by a number which states the chapter and number of which order the figure, table, equation or listing arrives in. As an example Figure 2.3 means that the figure is the third figure of chapter 2.

Below follows lists of the abbreviations and mathematical expressions, which has been used through the report.

Abbreviations

Abbreviation	Description
3GPP	3rd Generation Partnership Project
ADC	Analog to Digital Converter
ASIC	Application Specific Integrated Circuit
AWGN	Additive White Gaussian Noise
BEC	Backward Error Correcting
BPSK	Binary Phase Shift Keying
BW	Bandwidth
CB	Code Block
CLB	Configurable Logic Block
CP	Cyclic Prefix
CPLD	Complex Programmable Logic Device
CQI	Channel Quality Index
CRC	Cyclic Redundancy Check
DAC	Digital to Analog Converter
DFT	Discrete Fourier Transform
DSP	Digital Signal Processing
ECC	Error Correcting Code
eNodeB	Basestation notation in LTE
FEC	Forward Error Correcting
FFT	Fast Fourier Transform
FIFO	First In First Out
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
GMII	Gigabit Media Independent Interface
GUI	Graphical User Interface
HARQ	Hybrid Automatic Repeat Request
IFFT	Inverse Fast Fourier Transform
ISI	Inter Symbol Interference
IWS	Inter Window Shuffle
LLR	Log Likelihood Ratio
LSB	Least Significant Bit
LTE	Long Term Evolution
LUT	Lookup Table
MAC	Medium Access
MACC	Multiply Accumulate
MAP	Maximum A posteriori Probability
ML	Maximum Likelihood
MMS	Multimedia Messaging Service
MMSE	Minimum Mean Square Estimator
MSB	Most Significant Bit
MSE	Mean Square Estimator
NSC	Non Systematic Convolutional
OFDM	Orthogonal Frequency Division Multiplexing
PAPR	Peak to Average Power Ratio
PCCC	Parallel Concatenated Convolutional Codes

Continued on next page

Abbreviations – continued from previous page

Abbreviation	Description
PSK	Phase Shift Keying
QAM	Quadrature Amplitude Modulation
QPP	Quadratic Permutation Polynomial
RAM	Random Access Memory
RB	Resource Block
ROM	Read Only Memory
RSC	Recursive Systematic Convolutional
SC-FDM	Single Carrier - Frequency Division Multiplexing
SDR	Software Defined Radio
SMS	Short Message System
SNR	Signal to Noise Ratio
SOVA	Soft Output Viterbi Algorithm
TB	Transport Block
UE	User Equipment
UHD	USRP Hardware Driver
USRP	Universal Software Radio Peripheral
VRT	Vita Radio Protocol
ZF	Zero Forcing
ZPU	Softcore Processing Unit (designed by Xilinx)

Mathematical Notations

Expression	Description
(k)	Indexing over the discrete time domain
(n)	Indexing over the discrete frequency domain
$m(k)$	Binary message signal to be transmitted
$a(k)$	Encoded message signal to be transmitted
$b(k)$	Sequence of bit mapped symbols, which should be distributed onto OFDM subcarriers and transmitted
$s(k)$	Baseband signal output of the transmitter
$\tilde{s}(k)$	Received baseband signal
$\tilde{s}'$	Interleaved received baseband signal
$\tilde{s}_e$	Equalized received baseband signal
$\tilde{b}(k)$	Sequence of modulated symbols, gathered from OFDM demodulated subcarriers
$\tilde{a}(k)$	Received encoded bit sequence
$\tilde{m}(k)$	Received, demodulated and decoded message signal
$h(k)$	Discrete channel model
$c(k)$	Estimated channel taps
n_{AWGN}	AWGN noise of the channel
τ_k	Path delay
α_k	Complex fading coefficient for a given τ_k
pdf_{rayleigh}	The pdf of a Rayleigh channel
pdf_{Rician}	The pdf of a Rician channel
$\text{CRC}_\text{A}(D)$	CRC attached to the TB
$\text{CRC}_\text{A}(D)$	CRC attached to the CB
N_{suffix}	The total length of the array described by the $suffix$
δ_{suffix}	Denotes the throughput decrease of the function described by the $suffix$

Contents

Chapter 1

Introduction

The demand for faster and better communication systems is higher than ever. In it's infancy, mobile devices were meant to be used for phone calls, which were a basic communication method. Later SMS[1] arrived, enabling users to send fast messages. This then evolved into MMS[2], video calls and then finally now users have access to internet services like mails and browser surfing. Lately the interest in streaming full length movies on their mobile device has increased, making higher demand for available data traffic to the common user. From 2009 to 2014 it is estimated that data traffic will grow 39 times [1].

The increased demand leaves a problem since the available bandwidth in the air remains the same. To better utilize the available spectrum, calculation heavy en-/de-coding and (de)modulation algorithms are put to use. However, with the increase in calculation complexity follows an increase in power consumption and hence the need for faster data processors. This is undesirable since the equipment which has the wireless system implemented is often portable and therefore driven by battery power.

One of the high-end standards of telecommunication of today is LTE[3], which is proposed by the 3GPP[4]. The LTE system offers advantages like low complexity, high data rates[5] and variable bandwidth, which makes it the chosen communication type of today. The LTE system is a evolving system, where each new wave of LTE equipment increases the throughput.

The evolution of LTE creates a need for better telecommunication equipment. Here the FPGA[6] system is a good candidate for the future, since it is able to deliver the needed performance to support LTE but also provides the reconfigurability which future proofs the design [3].

[1] Short Message System
[2] Multimedia Messaging Service
[3] Long Term Evolution
[4] Third Generation Partner Project
[5] up to 300 Mbps [2]
[6] Field Programmable Gate Arrays

With the release of the first wave of LTE networks by TDC[7] in October 2011 the evolution began, although the technology is advancing fast there are still implementation challenges, e.g. MIMO utilization, before the theoretical peak data rates can be met. The everlasting challenge is the coherency between battery power, processing power, equipment size and cost.

In order to achieve better battery efficiency, speed up calculation time of complex algorithms and yield more processing power, technologies such as ASICs[8] or FPGAs are used. To fully utilize the power of an FPGA, it is necessary for the designer to understand the algorithms of the given system. This knowledge enables the designer to analyse and define which parts of the functionalities that are tractable to implement on the FPGA, or an ASIC[9], when considering the requirement of the system.

1.1 Problem statement

The introduction states that the utilization of ASIC/FPGAs are inevitable due to the increasingly complex algorithms used in wireless communications. Since the base station of a wireless system often runs on a stationary power source, the focus will be on the UE[10]. The following questions has to be answered to be able to implement a wireless communication system.

> ***Which functionalities are to be utilized when a wireless communication is to achieve high throughput, high robustness and offer a tractable solution of implementation.***

As mentioned, LTE is the standard of today which offers the highest data throughput. Due to this fact, the specifications of LTE will be the base of this report. This leads to the following sub goals:

- What is the motivation for using the different functionalities in LTE?
- How is it possible to create an testbed for a software radio which can utilize the power of ASIC/FPGA's?
- Which of the physical layer functions are tractable to implement in hardware?
- How is it possible to implement the desired functionalities in hardware and what is the gain?

[7] Former: Tele Danmark
[8] Application Specific Integrated Circuits
[9] Application Specific Integrated Circuits
[10] User Equipment (ie. mobile phone)

Chapter 2

Physical Layer Structure

The physical layer of a communication system handles all the data processing of the bits which should be transmitted/received, such as (de)coding, detection and transmission. A basic physical layer communication chain can be seen on Figure 2.1. An ideal reference for choice of functionalities to be used in the physical layer which is to be analysed is LTE since it is the high-end standard of today, when regarding mobile communication throughput. The focus of this chapter is to understand the functionalities of the physical layer.

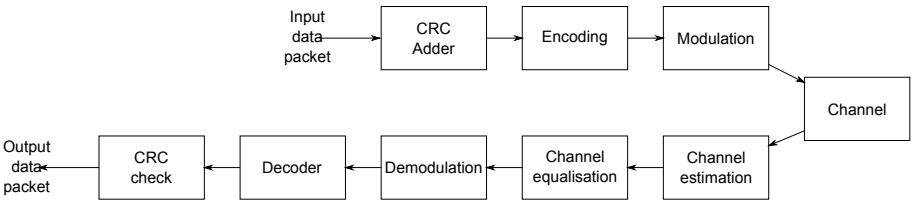


Figure 2.1: A basic radio communication link of a physical layer.

In this chapter the blocks seen in Figure 2.1 will be described in the same order as they are applied to the information sequence, i.e: the first thing that will be discussed is the CRC[1] and hereafter the Turbo Encoding.

When discussing the different functions of the transmitter side, the inverse function will be described right after, e.g: when the Turbo Encoder has been described, the Turbo Decoder will be described next section, even though it actually belongs to the receiver part.

The data input to the physical layer will be defined as a TB[2] and can be of any length. This data is the actual packet which the above layers wants to transmit, whereas a CB[3] is a small slice, with predefined maximum length, of the transport block which is subject to the encoding of the physical layer. The

[1] Cyclic Redundancy Check
[2] Transport Block
[3] Code Block

first step in the physical layer is to add an error detection code which enables the receiver to see if the given CB or TB has been received correctly, this is done in the CRC block.

2.1 CRC

By adding a CRC to the CB it is possible to check the received CB for errors. If errors are present the UE can request a retransmission from eNodeB[4] of the specific CB (BEC[5]). Finally, a CRC is attached to the complete TB, to ensure that the packet actually has been correctly received. Since the added checksum is redundant information it will lower the throughput, however it is desirable to add since it is a fast way to estimate if the digital packet has been received correctly.

The shorter the length CB the less amount of bits are needed to be retransmitted on erroneous packet reception. Furthermore, the complexity of some of the other physical layer functionalities are reduced, e.g. the Turbo Decoder. However, a longer CB results in an increased effectiveness of the Turbo Decoder's error correcting capabilities [4].

The CRC code sequence is a binary sequence, which, when added to the tail of the bit sequence, results in the fact that that there are no remainder when dividing the bit sequence with the CRC polynomial [5]. It is encoded by adding the remainder of the modulus 2 division as the redundant information. The decoding process divides the received sequence with the polynomial and if the result is zero the packet is accepted [5].

In LTE, the TB is split up in CBs if the TB exceeds the length 6020 bits, which has been shown to give an effective trade-off between Turbo Code complexity and error correcting capabilities [6]. The two different CRC polynomials in the LTE standard are denoted as:

- $\mathrm{CRC_A}$: Which attaches a 24 bit CRC on the TB, which can be of infinite length. The polynomial is seen in Equation 2.1.
- $\mathrm{CRC_B}$: Which attaches a 24 bit CRC on the CB, which are a part of the TB of max length $N_{\mathrm{CB\_max}} = 6044$ (6020 without added CRC) [6]. The polynomial is seen in Equation 2.1.

$$\begin{aligned}\mathrm{CRC_A(D)} = [&D24 + D23 + D18 + D17 + D14 + D11 + D10 + ... \\ &D7 + D6 + D5 + D4 + D3 + D + 1] \qquad (2.1)\end{aligned}$$

$$\mathrm{CRC_B(D)} = [D24 + D23 + D6 + D5 + D + 1] \qquad (2.2)$$

The CRC's are attached to the TB as seen in Figure 2.2.

[4] Basestation notation in LTE
[5] Backward Error Correction

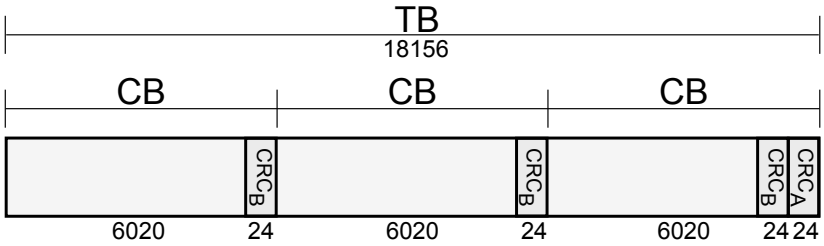


Figure 2.2: This figure illustrates how a desired TB of 18156 bits (including a 24bit CRC) for the TB are split into smaller CB's for which each has its own 24 bit CRC

Even though the added check value are redundant information it does not have a huge impact on the throughput. Given the length of the CB of 6048, with added CRC, it only results in decrease in throughput of:

$$\delta_{\text{CRC}} = \frac{N_{\text{CRC}}}{N_{\text{CB}}} = 0.39\ \%$$

2.2 Turbo code

Turbo codes, or PCCC[6], are a FEC[7] code. This means that it adds redundant information to the bit stream, which is used to help the receiver detect and correct incorrectly received bits. Turbo codes excel because of high robustness and the fact that they can come as close to the Shannon limit as 0.7 dB [7].

It is well known that NSC[8] code has a very good performance at high E_b/N_0 and a systematic code will be better at low E_b/N_0 [8]. However, at high code rates the RSC[9] code will outperform a NSC code at any E_b/N_0 [7]. The decoding complexity of an RSC code increases exponentially with the constraint length v[10]. An high complexity decoding process increases the resource usage of the system. Research has shown that parallel low low constraint length RSC codes can obtain the same error correcting capabilities as a high constraint length convolutional code, however the reduced constraint length of the low weight codes reduces the complexity of the decoding process significantly [7]. The Turbo Encoder consists of parallel RSC encoders, separated by interleavers, which generates the parity bits [7], as illustrated on Figure 2.3. In this section the input data to the Turbo Encoder will be denoted as U, the parity bit as Z and interleaving will be illustrated by an apostrophe.

The RSC encoders operate in parallel, instead of operating on each others outputs as would be the case for serial convolutional codes. Even though the RSC encoders operates on the same sequence, the inputs to the encoders are relatively uncorrelated, due to the interleaver. There are two main reasons for

[6]Parallel Concatenated Convolutional Codes
[7]Forward Error Correcting
[8]Non Systematic Convolutional
[9]Recursive Systematic Convolutional
[10]Number of delay elements in the code polynomial

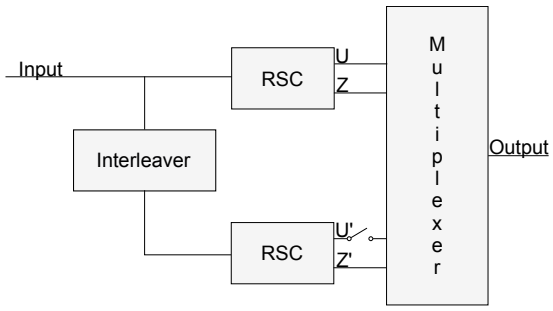


Figure 2.3: An example of a turbo encoder created by two RSCs connected by an interleaver ending up in three main outputs, thus making it a rate 1/3 encoder. On the sketch, the systematic output U' of encoder 2 have the possibility to be connected, which makes it an rate 1/4, this can be useful to obtain information about the tailing process, which will be described later. The multiplexer ensures that the data is organised in the desired order.

using the interleaver: The first is to increase the probability to create a high weight codeword and the other is to make the different codewords uncorrelated with each other, which increases the gain by the information exchange.

To decode the parallel encoding scheme, the Turbo decoder looks as depicted in Figure 2.4 of where it is clear that it uses RSC decoders in a serial connection, where the last encoder receives the information from the first.

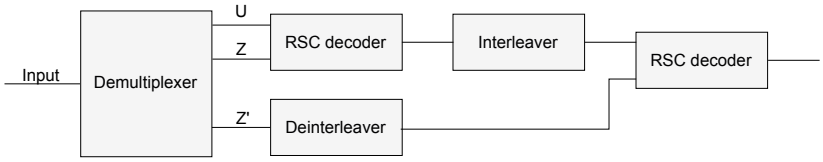


Figure 2.4: The principle behind the decoder for the parallel concatenation scheme of the Turbo encoder.

As seen on Figure 2.4, the result from the first decoder are interleaved, which ensures that the output of the first decoder, mimics the systematic input which was given to the second encoder [7]. To increase the performance given by the information exchange, the output of decoder one has to be a weighted soft decision bit such as the LLR[11]. The LLR describes the likelihood of the bit as seen in equation 2.3, a more detailed reason for this notation can be found in Chapter B. For a summed up description of some of the possible RSC decoder algorithms, refer to Section 2.2.1.

[11]Log Likelihood Ratio

$$\mathrm{LLR}\left(\tilde{a}(k)\right) = \mathrm{Log}\left(\frac{\mathrm{P}\left(\tilde{m}(k) = 1|\ \tilde{a}(k)\right)}{\mathrm{P}\left(\tilde{m}(k) = 0|\ \tilde{a}(k)\right)}\right) \tag{2.3}$$

Where:

- $\tilde{m}(k)$ is the decoded received message signal
- $\tilde{a}(k)$ is the received encoded bit sequence
- $\mathrm{P}\left(\tilde{m}(k) = 1|\ \tilde{a}(k)\right)$ are the a-posteriori probability of the message bit $\tilde{m}(k)$

The decoding scheme in Figure 2.4 is not optimal, since the first decoder only uses a fraction of the redundant information available [7]. As stated earlier in this section, the interleaver makes the two parity bits uncorrelated, which means that the decoder can benefit from a closed loop connection, as seen on Figure 2.5.

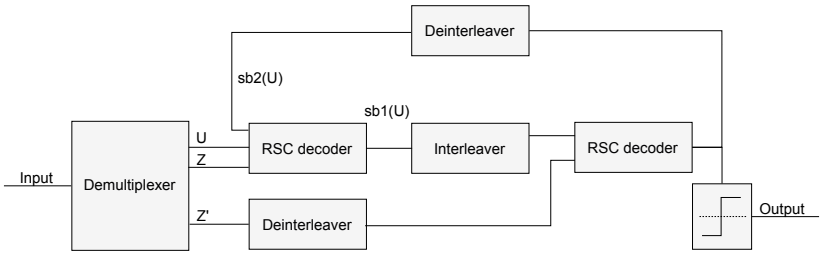


Figure 2.5: A closed loop Turbo decoder scheme which exchanges extrinsic information between the RSC decoders.

It has been shown, using a early stopping algorithm, that the average Turbo Decoder complexity can be substantially lowered without impact on the performance[9]. Furthermore it is shown that two early stopping algorithms work in compliment with each other. There are different options for a two level early stopping algorithm. Using hard decision, soft decision or a CRC. The latter is chosen for the project seeing that two level hard and soft decisions can result in erroneous decisions of otherwise correctly decoded bits and also that the CRC offers a greater reduction in average decoding complexity [10].

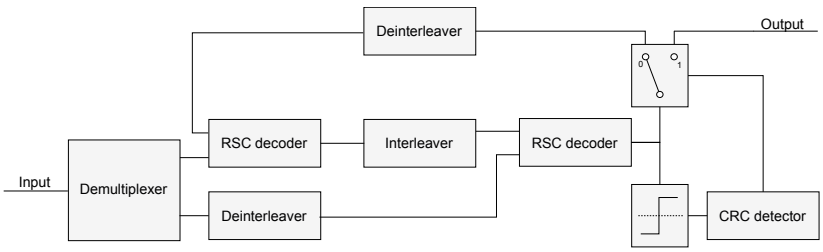


Figure 2.6: A closed loop Turbo decoder scheme with an added early stopping algorithm using the CRC on the CB.

As Figure 2.6 illustrates, the CRC decoder is implemented after each Turbo Decoding iteration. It operates on the hard decision bits of the Turbo Decoder, compares the newly calculated CRC with the received one and if they match, it will tell the Turbo Decoder that no further iterations are needed. If however, the CRC does not match, the Decoder will run a preset number of iterations, before accepting failure and requesting the MAC[12] layer to request a retransmission of the CB.

2.2.1 RSC - Decoder Algorithms

Before going into the RSC decoding a quick insight into the encoding process is required: One way to perceive the coding process RSC code is as a FSM[13], where the transition bits decide the code word for the current bit, and the next state decides the point of reference for the next bit. This way of perceiving the encoding process can also be illustrated through a Trellis diagram as shown on Figure 2.7. The encoder always starts in the state where all the memory registers, seen on Figure 4.2 are zero, since this increases the performance of the decoder.

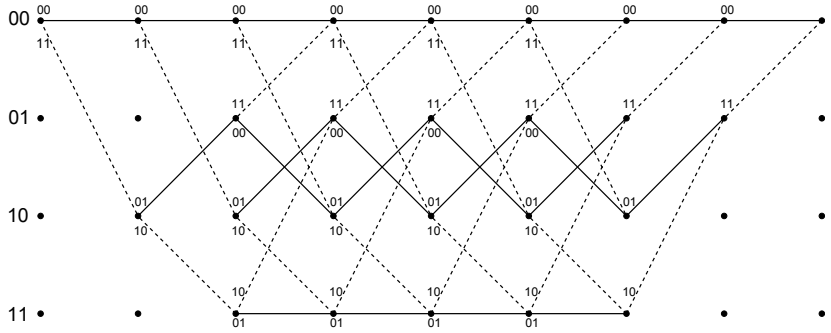


Figure 2.7: A trellis diagram which can be used to find the code word for the input bit by looking at the the current state and the input bit. The dashed lines illustrates a transition happening with a zero as input and the full line happens when it gets a one as an input [11].

The RSC decoders utilizes the trellis structure, shown in Figure 2.7, by calculating the likelihood of each transition and combining it to the likelihood of ending up in each state. Different algorithms to decode an RSC code has been proposed, where-as four have been chosen to be further described. Below is a list of items, summarising the chosen algorithms along with some pro's and con's. The summary is based on the findings in the book "Turbo Codes" [4].

- Viterbi Algorithm

Principle Uses ML[14] to maximize the probability $P_r(r \mid c)$, where r is the received bit and c is the transmitted bit. This is also equal to

[12] Medium ACcess
[13] Finite State Machine
[14] Maximum Likelihood

minimizing the euclidean distance between the received signal and the possible transmitted signals.

Pro's Has a low complexity, due to the lack of a backwards recursive part.

Con's The Viterbi algorithm only makes hard decisions, which causes it to lose some accuracy at multi stage decoding.

- SOVA[15]

Principle Also utilizes ML to evaluate the hard decision as the Viterbi algorithm, but uses the LLR to get the soft bit evaluation $\log\left(\frac{P_r\{c_t=1|r_1^\tau\}}{P_r\{c_t=0|r_1^\tau\}}\right)$.

Pro's Can utilize soft bits for optimized accuracy at multi stage decoding. Furthermore SOVA has a relatively low complexity compared other soft bit decoders as seen on Table 4.2.

Con's Can be up to twice as complex as the Viterbi algorithm.

- BCJR

Principle This algorithm uses MAP[16] to compute the LLR of $\log\left(\frac{P_r\{c_t=1|r\}}{P_r\{c_t=0|r\}}\right)$. This is done by calculating the 3 intermediate probabilities; the probability of transitioning from one state to another γ, the forward recursive probability α and the backward recursive probability β.

Pro's Though the algorithm is computationally heavy, the result of the algorithm is very robust.

Con's The complexity of the algorithm is high, which can be ascribed the use of exponential functions in the calculations as seen in Table 4.2.

- Max-Log-MAP

Principle This algorithm is a variation of the MAP decoder. As with SOVA the algorithm calculates the LLR of each branch metric. The Max-Log-MAP calculates the logarithmic α, β, and γ and uses these values to evaluate the LLR.

Pro's Compared to the BCJR algorithm the Max-Log-MAP is less complex than the MAP algorithm, because the multiplications of the MAP algorithm is converted to additions in the Max-Log-MAP.

Con's The reduced complexity of Max-Log-MAP, comes at the price of accuracy.

To fully use the potential of Turbo codes they should be connected in a closed loop as in Figure 2.5, which means that the BCJR or Max-Log-Map algorithms are the only possible means of implementation, due to the fact that they are very robust compared to the Viterbi and SOVA algorithm.

2.2.2 Rate Matching

The Turbo encoder in LTE is a rate $1/3$ as the one seen in Figure 2.3, with the RSC encoder which is depicted in Figure 2.8.

[15]Soft Output Viterbi Algorithm
[16]Maximum A posteriori Probability

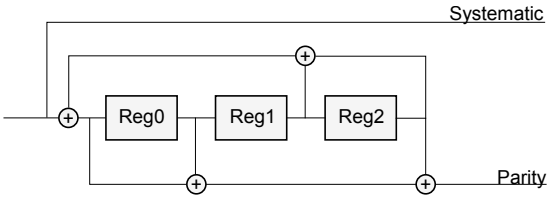


Figure 2.8: The RSC used in the LTE turbo encoder. As illustrated, it consists of three one-bit memory registers, connected with XOR additions.

Using an FEC code of rate 1/3 leads to an reduction in bitrate of $66.\overline{66}$ % since 3 bits has to be transmitted for each information bit in the sequence $m(k)$, as seen in Equation 2.4.

$$\delta_{\text{TC}} = \frac{N_{\text{Redundant bits}}}{N_{\text{Transmitted bits}}} = \frac{2}{3} = 66.\overline{66} \quad [\%] \quad (2.4)$$

At low SNRs[17] this reduction is compensated for by the error correcting capabilities of the code, since this will increase the throughput compared to if no FEC code was present. LTE offers different coderates by puncturing or reusing the data, which can be used together with changing the modulation order, to maximize throughput at a given SNR. To ensure granularity of the output, a rate matching algorithm is defined in the LTE standard. The rate matching algorithm interleaves the systematic bit sequence and spreads the parity bits such that the average amount of code bits when looking at small parts of the codeword are equal.

Firstly, the output data of the Turbo encoder will be inserted in a matrix as [12, sec. 5.1.4]:

$$b_k^j = \begin{bmatrix} 0_{0:I-1} & U_{0:K-1} & U_K & Z_{K+1} & U'_K & Z'_{K+1} \\ 0_{0:I-1} & Z_{0:K-1} & Z_K & U_{K+2} & Z'_K & U'_{K+2} \\ 0_{0:I-1} & Z'_{0:K-1} & U_{K+1} & Z_{K+2} & U'_{K+1} & Z'_{K+2} \end{bmatrix} \quad (2.5)$$

Where

- K is the length of the bit stream m.
- $0_{0:I-1}$ illustrates that each of the rows are zero padded in the beginning until the length of the array $N_{array} \text{mod } 32 = 0$.
- $I = 32 - ((K+4) \text{mod } 32)$.
- The bits from K to $K+2$ are the tail bits which have been used to terminate the trellis. As can be seen; both of the systematic output's tailbits are included, this is because these are not equal since the end states of the encodes are different due to the interleaver.

The three rows of b are interleaved independently, were b^0 and b^1 are interleaved with a block interleaver, described by the algorithm below:

[17] Signal to Noise Ratio

1. Write the data row wise into $R \times 32$ matrix, with R being the number of rows, found as $R = {}^{N_{array}}/_{32}$.
2. Column-permutate the matrix according to the polynomial in Equation 2.6, which takes the bit reversed column numbers.
3. Read the data out column wise

$$\begin{aligned} P = &[0\ 16\ 8\ 24\ 4\ 20\ 12\ 28\ 2\ 18\ 10\ 26\ 6\ 22\ 14\ 30\ ... \\ &1\ 17\ 9\ 25\ 5\ 21\ 13\ 29\ 3\ 19\ 11\ 27\ 7\ 23\ 15\ 31] \end{aligned} \tag{2.6}$$

b^2 are interleaved using the polynomial seen in equation 2.7.

$$\Pi_{b^2}(k) = \left(P\left(\left\lfloor \frac{k}{R} \right\rfloor \right) + 32\,(k \bmod R) + 1 \right) \bmod N_{array} \tag{2.7}$$

After the three rows has been interleaved, they are collected to the bit stream $a(k)$ as in Equation 2.8. It is to be noted that the padding bits are to be skipped when collecting the data.

$$a(k) = \begin{cases} b_k^0 & \text{for } k \in (0, 1, 2, ..., K-1) \\ b_k^1 & \text{for } k \in (K+0, K+2, K+4, ..., 3K-2) \\ b_k^2 & \text{for } k \in (K+1, K+3, K+5, ..., 3K-1) \end{cases} \tag{2.8}$$

The rate matched output sequence are obtained by starting with index

$$k_0 = R\left(2 \left\lceil \frac{K}{8R} \right\rceil rv_{idx} + 2 \right) \tag{2.9}$$

Where $rv_{idx} \in (0, 1, 2, 3)$ are the transmission number (when using HARQ[18] the first attempted transmission is 0 and a max of 3 retransmissions are allowed).

The bits to be transmitted are in the array $a(k)$ and the amount of data is read out such that $k \in (k_0, k_0+1, k_0+2, ..., k_0 + \lfloor {}^{3K+4\cdot 3}/_{G} \rfloor$, with G being the desired rate.

2.3 Interleaving

Interleaving is a function which takes the input sequence and shuffles the index sequence in order to either spread the indexed distance of each adjecent bit in the data array as far from each other as possible, or to create a new sequence which is as uncorrelated to the input sequence as possible.

Interleaving can, amongst other things, be used to counteract deep fades in a channel by increasing the indexed distance in the data array, between originally adjacent bits.

[18] Hybrid Automatic Repeat Request

	transmitted "sequence" T_{seq}	interleaving works!
a)	T_{seq} received	int_____ving works!
b)	T_{seq} interleaved $T_{seq}(I)$	ilnrevwsnegk!rio at
	T_{seq}' received	igl_____n ensrraiwt
	T_{seq}' received deinterleaved	int_rlea_ing w_r_s_

Table 2.1: An example of the idea of how interleaving helps spread the effects of deep fades in order to protect the information in the transmitted sequence. a) shows the sequence transmitted without the use of interleaving and b) shows the same transmission with the sequence interleaved, "transmitted" and then deinterleaved.

As seen in Table 2.1, interleaving increases the robustness of the data to fades by spreading the corrupted parts of the signal across the whole signal length such that deep fades will not distort a big subsequent part of the signal, but rather distort small parts of the signal, increasing the possibility of error corrections by the receiver. Furthermore, when implementing interleaving in the Turbo Codes it also increases the robustness of the data to noise due to the fact that the errors from one of the decoders also will be spread out when reaching the other [4].

There are many ways to design an interleaver algorithm. One of the more simple methods is called Block interleaving. This method uses a matrix of size $M \times N$ where the size of M and N is adjusted so that $M \cdot N$ equals the length of the input sequence L, this is illustrated below.

$$\tilde{s} = \{s_0\ s_1\ s_2\ s_3\ s_4\ s_5\ s_6\ s_7\ s_8\}$$

$$\begin{Bmatrix} s_0 & s_1 & s_2 \\ s_3 & s_4 & s_5 \\ s_6 & s_7 & s_8 \end{Bmatrix}$$

$$\tilde{s}' = \{s_0\ s_3\ s_6\ s_1\ s_4\ s_7\ s_2\ s_5\ s_8\}$$

The input sequence is read into the the matrix row-wise and the interleaver output is then read out column-wise, thus shuffling the indexes compared to the input. To further scramble the order of the bits, the rows can be permutated.

The LTE standard suggests the use of a contention-free QPP[19] interleaver[13] which is a interleaver type that uses a polynomial to calculate the interleaving process. This polynomial is illustrated below.

$$\Pi_{\text{Turbo Codes}}(k) = \left(f_1 \cdot k + f_2 \cdot k^2\right) \mod K$$

[19]Quadratic Permutation Polynomial

The variables f_1 and f_2 are integer values that correspond to the data length K. There are 188 defined data lengths in LTE, for which f_1 and f_2 are listed in a Look-up table in the standard [12].

Choosing a QPP type interleaver over other interleavers, is due to it's simplicity, with only to integers describing the permutation, while additionally providing contention free memory access. It requires little storage space while offering a high degree of parallelism with no loss in performance compared to contention free IWS[20] interleaver, which has higher requirements.[14][15]

2.4 Air Interface

In order to enable a bit stream to be transmitted through an antenna, it must first be modulated onto a specific carrier or array of carriers, that can convey the bit stream through the antenna. This section will focus on discussing the different modulation techniques used in LTE And the motivation of this.

It has been chosen to investigate the OFDM[21] modulation since this is used in LTE. OFDM has the advantage of increased symbol time, which is helpful in a multipath environment where fading occurs, while retaining the same throughput. Furthermore, it utilizes the spectrum more efficiently than single carrier modulation.

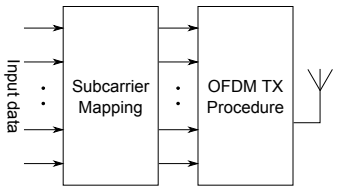


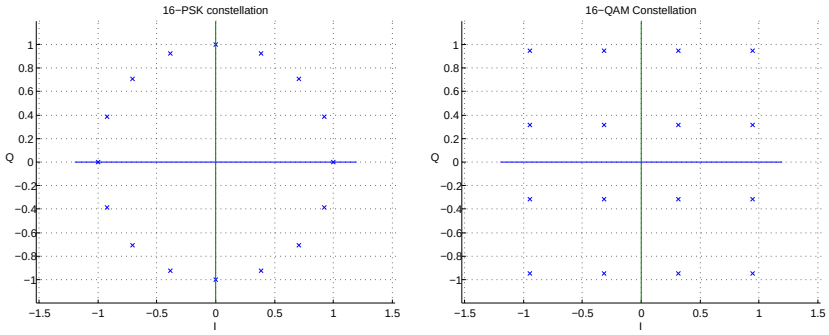
Figure 2.9: The order of precedence of the air interface defined in LTE.

OFDM makes use of several orthogonal sub-carriers, which carries bits which is mapped onto a desired constellation[22]. As seen in Figure 2.9 the binary message in mapped to the complex carrier using PSK[23] or QAM[24]. First the desired modulation schemes is explained, whereafter the basics behind OFDM is explained. Note that in this chapter, "symbol" defines the output of the constellation mapper and input to the constellation demapper, whereas "OFDM-symbol" is the term for the actual output of the transmitter and input to the receiver.

[20] Inter Window Shuffle
[21] Orthogonal Frequency Division Multiplexing
[22] Hereafter refereed to as bit-modulation, so it is not to confuse with the OFDM modulation
[23] Phase Shift Keying
[24] Quadrature Amplitude Modulation

2.4.1 Subcarrier Mapping

PSK and QAM are two different ways of mapping the transmitted bits onto a symbol, defined as a complex number. An example of the two schemes are seen in Figure 2.10.



(a) PSK constellation. **(b)** QAM constellation.

Figure 2.10: An example og 16-PSK and 16-QAM constellation diagram.

PSK has the advantage of all symbols having the same envelope, however the euclidean distance between the symbols are smaller than in QAM, making it more susceptible to noise. The QAM constellation has the disadvantage that, the receiver having to know the maximum amplitude of the received symbol, since the decoding is dependant on the amplitude, whereas PSK only depends on the phase.

The euclidean distance between each symbol for a 16-PSK modulation $d_{16\text{PSK}}$ can be found as in Equation 2.10 and for 16-QAM modulation as in Equation 2.11.

$$d_{16\text{PSK}} = 2\sqrt{E_s}\sin\left(\frac{\pi}{16}\right) \tag{2.10}$$

$$d_{16\text{QAM}} = 2\sqrt{\frac{E_s}{10}} \tag{2.11}$$

The ratio between the two modulation types, assumed that the $\frac{E_s}{N_0}$ is equal, is 1.62 which in dB is $20\log(1.62) = 4.19$ dB, as is seen in Equation 2.12.

$$ratio\left(\frac{16\text{QAM}}{16\text{PSK}}\right) = \frac{2\sqrt{\frac{E_s}{10}}}{2\sqrt{E_s}\sin\left(\frac{\pi}{16}\right)} = 1.62 \tag{2.12}$$

This means that 16-QAM will require 4.19 dB less signal power to achieve the same signal error rate as 16-PSK. Therefore QAM has been chosen for further analysis.

The different constellation diagrams of the available modulation schemes in LTE are; BPSK[25], QAM, 16QAM and 64QAM, which can be found illustrated

[25] Binary PSK

in Figure B.1, B.3, B.4 and B.5 accordingly, in Appendix B. The used constellations are grey encoded, which means that the hamming distance between each subsequent adjacent symbol is no more than 1. The low Hamming distance greatly decreases the complexity of the de-mapping algorithm as will be seen in this section.

The modulation of the signals are very simple, since it can be done by lookup-tables or as illustrated in the equations 2.13 to 2.15.

$$s_{\mathrm{BPSK}}(k) = (-1)^{|m(k)-1|} \tag{2.13}$$

$$s_{\mathrm{QAM}}(k) = \left((-1)^{|m(2k)-1|} + \mathrm{j}(-1)^{|m(2k+1)-1|}\right) \cdot 1/\sqrt{2} \tag{2.14}$$

$$\begin{aligned} s_{16-\mathrm{QAM}}(k) = (-1)^{|m(k+3)-1|} \cdot \left(\frac{3m(k+1)}{\sqrt{10}} + \frac{|m(k+1)-1|}{\sqrt{10}}\right) + ... \\ \mathrm{j}(-1)^{|m(k+2)-1|} \cdot \left(\frac{3m(k)}{\sqrt{10}} + \frac{|m(k)-1|}{\sqrt{10}}\right) \end{aligned} \tag{2.15}$$

The 64-QAM expression is obtained by expanding the principle of 16-QAM in equation 2.15.

When decoding a bit, it is assumed that it is affected by an AWGN[26] channel, described in Appendix B. When using an AWGN channel with variance σ_n^2, the received value probability of the bit can be described by the use of the normal distribution, as seen in equation 2.16, where μ is the value of the transmitted symbol.

$$\mathrm{P}\left(\tilde{a}(k) = \mu \mid \tilde{b}(k)\right) = \frac{1}{\sqrt{2 \cdot \pi \cdot \sigma_n^2}} \cdot \exp\left(-\frac{(\tilde{b}(k) - \mu)^2}{2 \cdot \sigma_n^2}\right) \tag{2.16}$$

Where, $\tilde{b}(k)$ is a sequence of received symbols gathered from OFDM subcarriers.

In section 2.2 it is described that the Turbo Decoding algorithms uses a soft valued bit, describing the probability of the bit being correct, as input. As seen in Appendix B the soft valued bit are calculated based on the MAP decision rule, showed in Equation 2.17.

$$\frac{\mathrm{P}(\tilde{a}(k) = \mu_1 \mid \tilde{b}(k))}{\mathrm{P}(\tilde{a}(k) = \mu_0 \mid \tilde{b}(k))} = \frac{\mathrm{P}(a(k) = \mu_1)}{\mathrm{P}(a(k) = \mu_0)} \tag{2.17}$$

Since the probability of the bit is described using the exp function, it can be simplified by using the logarithm, which leads to the LLR, which leads to

[26] Additive White Gaussian Noise

the calculation of the soft bit as seen in Equation 2.18.

$$\begin{aligned}\log\left(\frac{P(\tilde{a}(k)=\mu_1 \mid \tilde{b}(k))}{P(\tilde{a}(k)=\mu_0 \mid \tilde{b}(k))}\right) &= \log\left(\frac{\frac{1}{\sqrt{2\cdot\pi\cdot\sigma_n^2}}\cdot\exp\left(-\frac{(\tilde{b}(k)-\mu_1)^2}{2\cdot\sigma_n^2}\right)}{\frac{1}{\sqrt{2\cdot\pi\cdot\sigma_n^2}}\cdot\exp\left(-\frac{(\tilde{b}(k)\mu_0)^2}{2\cdot\sigma_n^2}\right)}\right) \\ &= \log\left(\exp\left(-\frac{(\tilde{b}(k)-\mu_1)^2}{2\cdot\sigma_n^2}\right)\right) - ... \\ &\log\left(\exp\left(-\frac{(\tilde{b}(k)-\mu_0)^2}{2\cdot\sigma_n^2}\right)\right) \\ &= -\frac{(\tilde{b}(k)-\mu_1)^2}{2\cdot\sigma_n^2} + \frac{(\tilde{b}(k)-\mu_0)^2}{2\cdot\sigma_n^2} \end{aligned} \tag{2.18}$$

With the LLR of a single bit defined, it is possible to deduct a soft decision algorithm for the constellations defined in LTE. Research has shown that soft decoding can increase performance compared to hard decoding by up to 8.5 dB [16]. The next subsections will make a breif summary of the analysis in Appendix B which is based on [16].

2.4.1.1 BPSK soft-output algorithm

The demodulation of the bit-mapping are done by the algorithms given in Appendix B. The algorithms deducted in those sections are, however, fairly computational heavy since they are based on the LLR calculation seen in Equation 2.19

$$\mathrm{LLR}(\tilde{a}(k)) = -\frac{(\tilde{b}(k)-\mu_0)^2}{2\cdot\sigma_n^2} + \frac{(\tilde{b}(k)-\mu_1)^2}{2\cdot\sigma_n^2} \tag{2.19}$$

As seen in Equation 2.19 the algorithm uses two subtractions, 6 multiplications and one addition to calculate the result. Research has shown that Equation 2.19 can be simplified by only calculating the euclidean distance between the symbols [16].

$$\mathrm{LLR}(\tilde{a}(k)) = -\tilde{b}(k) \tag{2.20}$$

The simplification seen in Equation 2.20 does not lead to a performance loss compared to the theoretical calulations but they does lead to an increase in speed [16].

2.4.1.2 QAM soft-output algorithm

In the case of QAM, the problem can be split up so that the in phase and quadrature component describes a bit each. This means that the two soft bit outputs in Equations B.9 and B.10 can be described by the simplification in equation 2.21 and 2.22.

$$\mathrm{LLR}(\tilde{a}(2k)) = -\tilde{b}_{\mathrm{real}}(k) \tag{2.21}$$

$$\mathrm{LLR}(\tilde{a}(2k+1)) = -\tilde{b}_{\mathrm{imag}}(k) \tag{2.22}$$

2.4.1.3 16-QAM soft-output algorithm

As with QAM, it is possible to calculate each soft bit value using the LLR Equation 2.18. The four bits held by the 16-QAM constellation are denoted as illustrated by the expression in Equation 2.23.

$$\tilde{a}_{4k:4k+3} = [D_{I,1}, D_{Q,1}, D_{I,2}, D_{Q,2}] \tag{2.23}$$

Using the simplification on Equation B.11 to B.12 yields the result seen in Equation 2.24 and 2.25. Note that the calculation of $D_{I,k} = D_{Q,k}$ uses the in phase and quadrature component of $b(k)$ respectively.

$$D_{I,1} = -\tilde{a}(k) \tag{2.24}$$

$$D_{I,2} = |\tilde{a}(k)| - \frac{3}{\sqrt{10}} \tag{2.25}$$

2.4.1.4 64-QAM soft-output algorithm

The deduction and denotion of the 64-QAM is similar to the 16-QAM. The bit denotion are seen in equation 2.26 and the algorithm are seen in equation 2.27, 2.28 and 2.29.

$$\tilde{a}_{6k:6k+5} = [D_{I,1}, D_{Q,1}, D_{I,2}, D_{Q,2}, D_{I,3}, D_{Q,3}] \tag{2.26}$$

$$D_{I,1} = -\tilde{a}(k) \tag{2.27}$$

$$D_{I,2} = |\tilde{a}| - \frac{5}{\sqrt{42}} \tag{2.28}$$

$$D_{I,3} = \left| -|\tilde{a}| + \frac{5}{\sqrt{42}} \right| - \frac{3}{\sqrt{42}} \tag{2.29}$$

2.4.2 OFDM

OFDM is a broadband multicarrier modulation which performance excels in spectrum utilization compared to other modulation methods. Furthermore it is effective at mitigating ISI[27] due to long symbol times.

Single carrier transmission with a high symbol rate is very susceptible to the deep fades and ISI, which can occur in a multipath channel environment as described in Section 2.5. Channel estimation can be used to counteract the influence of these distortions, however this becomes increasingly difficult as the

[27] Inter Symbol Interference

desired data rate increases, due to the decreased symbol time.

To mitigate the effects of ISI, OFDM transmits multiple narrowband subcarriers, modulated by PSK/QAM, on the same broadband medium as it is the case with Frequency-Division multiplexing. OFDM utilizes the broadband spectrum to its fullest, by ensuring that each carrier is orthogonal to each other, e.g. that the current subcarrier lies in the nulls of the adjacent subcarriers frequency response. Orthogonality is obtained by setting the frequency spacing f_s to be equal to the reciprocal of the symbol time:

$$f_s = \frac{1}{t_{\text{symbol}}} \quad [\text{Hz}] \quad (2.30)$$

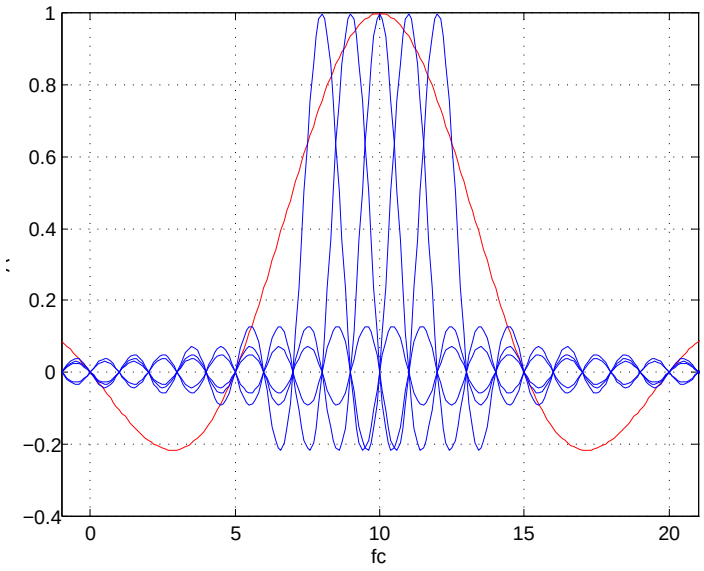


Figure 2.11: OFDM and single carrier transmission's utilization of the Frequency spectrum at a symbol rate of 5, where it is very clear that the OFDM utilizes the spectrum more efficiently while retaining the same datarate.

The multiple subcarriers of OFDM enables the system to have long symbol times, while retaining high data rates. Furthermore, OFDM utilizes the spectrum more efficiently than e.g. single carrier transmission at the same datarate due to its multiple orthogonal carriers, as shown on Figure 2.11.

The OFDM modulation can be handled by using a DFT[28] since this algorithm will transform the carriers between frequency an time domain, while retaining the frequency spacing. A DFT has a high calculation complexity which is not tractable to implement in a system with many subcarriers. The

[28] Discrete Fourier Transform

FFT[29] is a more efficient way of calculating the DFT which reduces the complexity from $O\left(N^2\right)$ for the DFT to $O\left(N\log\left(N\right)\right)$ for the FFT.

Despite OFDM's many advantages it is susceptible to carrier frequency offset from Doppler shift or from synchronization problems between transmitter and receiver. This frequency offset has a direct impact on the subcarriers SNR, but this can be counteracted by transmitting pilot signals in order for the receiver to synchronize with the transmitter[17].

To increase the robustness against ISI and other elements of the multipath fading channel the OFDM symbol is extended with redundant information in form of a CP[30] which, if defined in such a way that the length exceeds the expected delay spread, ensures that the data on each subcarrier will only be affected by a single channel coefficient as can be seen in Equation 2.31 [18].

$$\tilde{s}_k = Hs_k + n_k \tag{2.31}$$

The OFDM modulation chain is shown on Figure 2.12, and as is seen uses an (I)FFT[31] to ensure orthogonality.

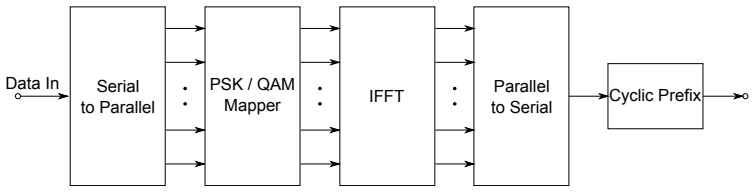


Figure 2.12: The OFDM modulation chain used in an LTE system.

An implementational disadvantage of the OFDM modulation is that it has a high PAPR[32], which increases the requirements to the linearity of the power amplifier of the transmitter. SC-FDM[33], also known as *Frequency spread* OFDM, uses multiple carriers to carry one symbol and hereby lowering the the PAPR compared to normal OFDM[19, 46]. This property is due to the fact that a single symbol is transmitted over several subcarriers, which results in fewer fluctuations. Tests has been made [20] which show that SC-FDM can obtain up to 10 dB lower PAPR compared to an OFDM transmission, when using QPSK.

Even though SC-FDM is more efficient, it chosen to only use this in the uplink since the basestation would not gain as much using excessively complex algorithms to save some power, when it is a resource that is easily available to the basestation. In this report, the focus is only on OFDM since the modulation chain of SC-FDM is an expanded version of OFDM as seen in Figure 2.13.

[29] Fast Fourier Transform
[30] Cyclic Prefix
[31] Inverse FFT
[32] Peak to Average Power Ratio
[33] Single Carrier - Frequency Division Multiplexing

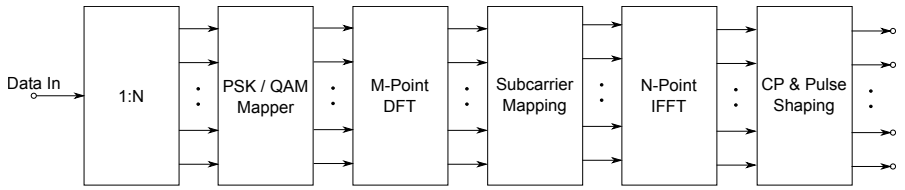


Figure 2.13: Illustration of the encoding chain of the SC-FDM.

The LTE standard states that the OFDM symbol time is $66.\overline{66}$ ms, which leads to a spacing of 15 kHz. To simplify the available resources for a transmission, the subcarriers are divided into RBs[34], which consists by 12 subcarriers over a slot duration of 0.5 ms[13, 39], where two adjacent slots is one sub-frame. This structure is illustrated on Figure 2.14.

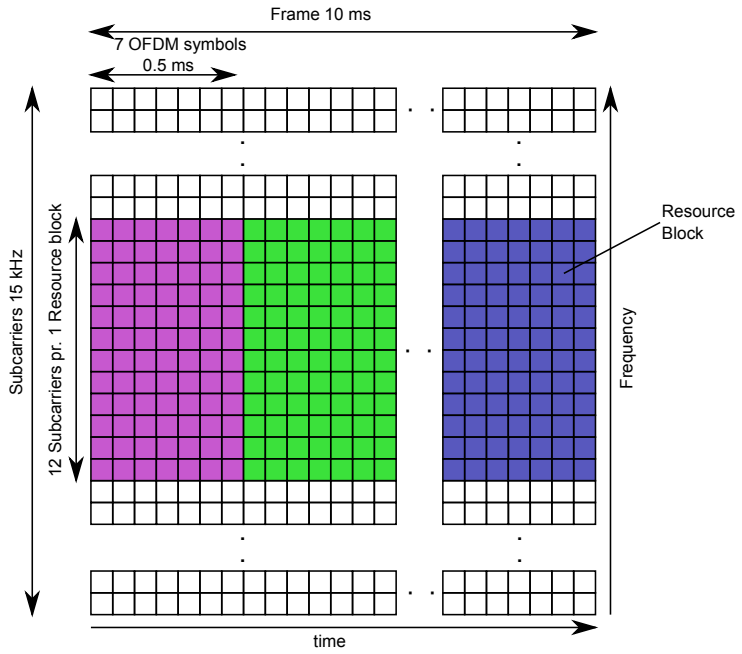


Figure 2.14: Illustration of a resource block in the LTE standard.

A radio frame, has a duration of 10 ms, consisting of 10 sub-frames.

LTE supports bandwidths of up to 20 MHz [13, 32] with 6 different choices of bandwidth which are stated in Table 2.2 together with how many subcarriers the standard dictates it is configured with.

LTE defines two CP lengths; the standard CP of 4,8 μs and the extended of 16,6 μs. The CP is created by copying first part of the sequence to the end of the symbol as illustrated on Figure 2.15 and the different lengths are coherent with either 7 or 6 OFDM symbols are placable within a timeslot.

[34] Resource Block

Transmission BW[35] (MHz)		1.4	3	5	10	15	20
Sub-frame duration		1.0 ms					
Sub-carrier spacing		15 kHz					
Sampling frequency (MHz)		1.92	3.84	7.68	15.36	23.04	30.72
FFT size		128	256	512	1024	1536	2048
Number of occupied sub-carriers		72	180	300	600	900	1200
CP length (μs)	Normal	4.69×6, 5.21×1					
	Extended	16.6					

Table 2.2: The table shows the utilization of the air medium depending on the required settings for the available bandwidth [21].

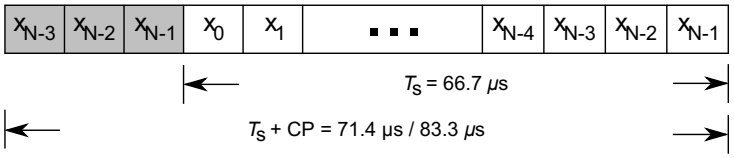


Figure 2.15: Illustration of how the CP is added to the OFDM symbol [11].

2.5 The Wireless Medium

This section will describe the used channel model with the purpose of analysing the medium and its impact.

A transmitted signal will before being received by the receiver propagate through a channel, where the signal is reflected through an infinite number of paths, closely spaced in time. However when modelling and handling the reflections they will be modelled by a filter with a finite number of taps, as in equation 2.32.

$$H(\tau) = \sum_{k=0}^{K-1} \alpha_k \delta(\tau - \tau_k) \tag{2.32}$$

With:

- $H(\tau)$ being the discrete channel model
- τ is the delay
- K being the number of taps
- τ_k is the path delay
- α_k is the complex fading coefficient for the given path delay τ_k

The environmental impact on the channel fading in the channel transfer function can be defined in two main types:

- Slow fading; which defines the signal fading typically caused by shadowing, e.g. obstruction of the direct path by the surroundings such as a hill. This type is of not interest, since the fading varies very slowly (across multiple symbols).
- Fast fading; which occurs when the signal reflects from objects, like trees or buildings, relatively close to the receiver and will vary within a symbol time.

Different models have been created of the fast fading filter tap power distribution, of which the mostly used are:

- Rayleigh fading, which assumes that there is no direct path between the transmitter and receiver and can especially be used to model densely build cities. The distribution is described by Equation 2.33 [22].

$$pdf_{\text{rayleigh}} = \frac{s}{\sigma^2} \cdot exp\left(\frac{-s^2}{2 \cdot \sigma^2}\right) \tag{2.33}$$

Where:

- pdf_{rayleigh} is the rayleigh pdf
- s is the transmitted signal
- σ^2 is the transmitted signal variance

And

- Rician fading; which are used when there are one path, typically a path in line of sight of the receiver, which are significantly stronger than the others. This distribution is described in equation 2.34 [22].

$$pdf_{\text{rician}} = I_0 \cdot \left(\frac{s^2 \cdot \lambda^2}{\sigma^2}\right) \cdot \frac{s}{\sigma^2} \cdot exp\left(-\left(\frac{s^2 + \lambda^2}{2 \cdot \sigma^2}\right)\right) \tag{2.34}$$

Where:

- pdf_{rician} is the rician pdf
- I_0 is the 0 order modified bessel function of the first kind.
- s is the transmitted signal
- λ is a noncentrality parameter (≥ 0 for $x > 0$)
- σ^2 is the transmitted signal variance

The different distributions can be used together with power delay profile models to successfully create a realistic model of the given environment. A power delay profile defines the intensity of a signal received as a function of the time dispersion.

With the channel problem properly described the channel estimation can be described.

2.5.1 Channel Estimation

The channel estimate is briefly described in order to give a glimpse of how the estimate is generated, before using it in the equalization. The channel estimate is found by transmitting some known reference signals on specific subcarriers in the frequency domain. These reference signals are spread out so that there is a offset of 6 subcarriers between each reference signal subcarrier as can be seen in Figure 2.16.

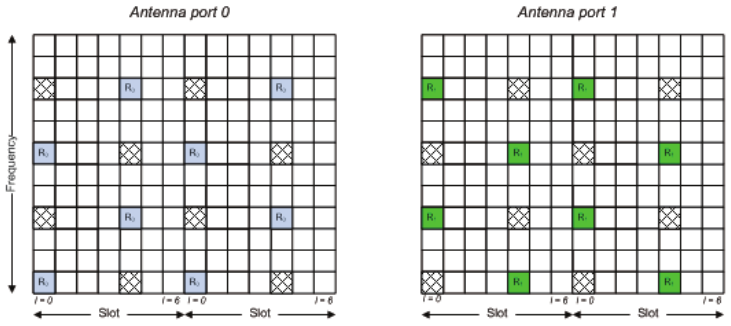


Figure 2.16: How the reference symbols are placed in the LTE with an two antenna transmitter system [21], as seen they are placed with a 6 subcarrier spacing and the symbol spacing is 2 with an offset of 4 subcarriers. Furthermore the opposite antenna of the one which is currently transmitting a reference symbol is silent for that specific subcarrier.[6]

The channel estimate across the whole BW is then found by interpolation of the transmitted reference signals. The channel estimate in the reference signal positions can be seen in equation 2.35 [23].

$$\hat{H}_{P,LS} = \frac{Y_P(n)}{X_P(n)} + z_{\text{AWGN}} \tag{2.35}$$

where

- z_{AWGN} is the AWGN noise of the channel
- $\hat{H}_{P,LS}$ is the least square estimation of the channel at the reference signal locations
- $Y_P(n) = H_P(n)X_P(n)$
- $H_P(n)$ is the channel gain at the reference symbol location
- $X_P(n)$ is the transmitted reference symbol

This will yield the channel estimate for the reference signal subcarriers. The complete channel estimate may then be obtained by using linear interpolation on the calculated estimate in equation 2.35 [24].

2.5.2 Channel Equalization

To help mitigate for the channel influence on the transmitted signal a equalizer can be utilized. This block utilizes a channel estimate to try and counteract some of the corruption of the channel. To accomplish this there are different types of equalizers that try to mitigate the channel corruption in different ways.

2.5.2.1 Zero Forcing Equalizer

The ZF[36] equalizer is a simple example of a type of equalizer. It assumes that the transmitted signal $s(k,n)$ is corrupted purely by ISI and seeks to equalize this as can be seen in equation 2.36.

$$\begin{aligned} \tilde{s}_e(k,n) &= C^H(n)\tilde{s}(k,n) = C^H(n)H(n)s(k,n) \\ C(n) &= \left(H^H(n)H(n)\right)^{-1} H^H(n) \end{aligned} \tag{2.36}$$

- $\tilde{s}_e(k,n)$ is the ZF estimator at the "n"-th subcarrier
- $H(n)$ is the channel gain at the "n"-th subcarrier
- $C(n)$ is the channel filter tap at the "n"-th subcarrier, which is calculated by using the pseudo inverse of $H(n)$

But since a normal channel environment also contains some AWGN the equation will look a bit different as seen in equation 2.37

$$\begin{aligned} \tilde{s}_e(k,n) &= C^H(n)\left(H(n)s(k,n) + z_{\text{AWGN}}\right) \\ &= C^H(n)H(n)s(k,n) + C^H(n)z_{\text{AWGN}} \end{aligned} \tag{2.37}$$

Since the filter taps $C(n)$ are not scaled to counteract the AWGN they risk amplifying it which is not desirable.

2.5.2.2 Minimum Mean Squared Error Equalizer

In order to expand the equalizer to incorporate the AWGN in the filter tap design the linear MMSE[37] equalizer is used.

This equalizer seeks to minimize the MSE[38] $E\left\{|s(k,n) - \tilde{s}(k,n)|^2\right\}$ thus designing the filter taps in the best way to counteract the channel.

The filter taps are as can be seen in equation 2.38 [25].

$$\begin{aligned} \tilde{s}_e(k,n) &= C^H(n)\left(H(n)s(k,n) + z_{\text{AWGN}}\right) \\ C(n) &= \left(H(n)\sigma^2_{s(k,n)}H^H(n) + \sigma^2_{z_{\text{AWGN}}}\right)^2 H(n)\sigma^2_{s(k,n)} \end{aligned} \tag{2.38}$$

[36] Zero Forcing
[37] Minimum MSE
[38] Mean Squared Error

The linear MMSE equalizer has a good performance while it has a relatively low complexity as opposed to the MAP equalizer which is very complex. Furthermore the complexity of the linear MMSE equalizer can be greatly reduced without too much loss of performance [26].

2.5.2.3 Turbo Equalizer

The equalizer can be incorporated into the turbo decoder in such a way that the equalizer estimate receives feedback a priori information from the decoder which in turn helps the equalizer to make a better estimate.

The principle behind the turbo equalizer is that the channel itself is viewed as part of the signal encoding, therefore expanding the turbo decoder to also utilize the equalizer. The information from the equalizer is decoded and then reencoded and fed back to the equalizer as extrinsic information thus allowing the equalizer to improve the estimate $\tilde{s}(n)$ of the received data. A block diagram of the turbo equalizer is shown in Figure 2.17.

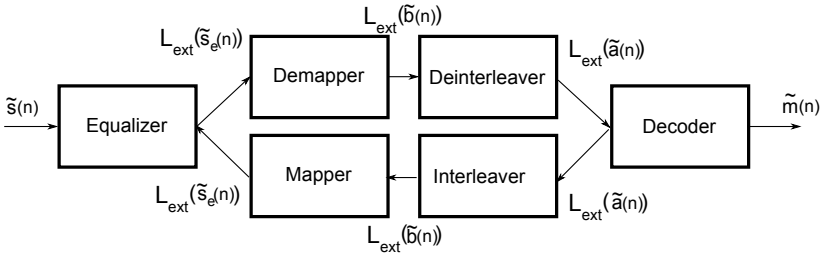


Figure 2.17: A block diagram of the turbo equalizer.

The decoder in the turbo equalizer is the same as in the turbo decoder, but in place of passing the extrinsic information to another decoder block it is fed back to the equalizer, which uses it to optimize the signal estimate $\tilde{s}(n)$[27].

When used in the turbo equalizer the linear MMSE equalizer gets feedback information from the decoder, which is used to improve the estimate $\tilde{s}_e(n)$. This changes equation 2.38 to take account for the feedback information as seen in equation 2.39.

$$\begin{aligned} \tilde{s}_{e,k} &= \mathrm{E}\{|s_k|\} + C_k^H \left(\tilde{s}_k - H_k \mathrm{E}\{|s_k|\}\right) \\ C_k &= \left(H_k \sigma_{s_k}^2 I_N H_k^H + \sigma_{n_k}^2 I_N\right)^2 H_k \sigma_{s_k}^2 I_N \end{aligned} \tag{2.39}$$

Here $\mathrm{E}\{|s_k|\}$ and $\sigma_{s_k}^2 I_N$ are both affected by the extrinsic information, which in turn optimizes the equalizer estimate $\tilde{s}_{e,k}$.

While the performance is a bit lower than that of the MAP equalizer the reduction in complexity more than makes up for it for the hybrid approximate MMSE LE equalizer[26]. This shows that a MMSE equalizer with lowered complexity can still perform sufficiently.

2.6 Physical Layer Structure - Summary

The functionalities in LTE's communication link has been analysed with respect why and how they are defined in the communication link. The high performance of LTE comes amongst other things from the highly robust Turbo codes, which has high error correcting capabilities and OFDM modulation which utilizes the avaliable spectrum very efficiently and can mitigate the effects of ISI using a CP.

The high performance of especially the OFDM and Turbo codes comes at the price of a high calculation complexity. High calculation complexity leads to either a high latency of the communication link, or the need of a high speed processor which uses a lot of power.

To further increase performance of the receiver, a Turbo equalizer can be utilized. The Turbo equalizer uses the soft output bits of the Turbo decoder through a feedback loop, which further increases the strain on the physical layers complexity.

It is clear that the Physical layer of a modern communication system holds a high complexity, but it does also have high constraints on the speed, latency and power consumption. On the basis of this, it is reasonable to analyse the Physical layer functionalities with respect to if they can gain from hardware acceleration.

Chapter 3

FPGA resources

This chapter addresses the choice of which hardware platform should be used for implementation. In order to make a viable decision of platform it is important to analyse the possibilities regarding available resources and interfaces.

The LTE technology is minded towards portable devices, such as cellphones, laptops or tablets. Due to the fact that the modern user wants a lot of computational power and battery power in his/hers portable devices, the implementation of the physical layer functionalities described in Chapter 2 is to be made with focus on both computation efficiency and power consumption.

To reduce battery usage while retaining a high calculation capability, the implementation could be created as a co-design between hardware and software. A way to do this is to utilize an FPGA or an ASIC which gives the programmer access to low level logic gate programming and further extended pipeline options than the one supplied by a processor using a Harvard architecture.

The choice of whether to choose an FPGA or an ASIC is a compromise between flexibility and power consumption: FPGA's are versatile due to the fact that it is possible to reconfigure the fabric[1], however they are often seen as power hungry devices compared[2] to the ASIC due to the structural design. ASICs can be very optimized with power consumption in mind, however they do not have the capability of being reconfigured as much as the FPGA.

A suitable platform for this project would be an USRP[3], which is a hardware platform created specifically for software radio projects by *Ettus Research LCC*. The USRP consists of an FPGA, with various interfaces, of which special interest are the 2 DACs and ADCs, which can be used to connect different available daughterboards. The converters are high speed (100 Msps) and has a 14-bit resolution, which is the minimum requirement if a high SNR should be achieved in modern communication systems [28].

[1] The part of the FPGA which is available for programming

[2] Often the increased power consumption of the FPGA are due to unused connections, which will still leak. This also means that a good fabric utilization results in power consumption close to an ASIC.

[3] Universal Software Radio Peripheral

The daugtherboards are RF front ends which allows the designer to utilize the USRP's resources with communication systems in mind. There are many different daughterboards available, currently enabling the designer to configure the transmission within the bandwidth ranging from 1 MHz to 5.9 GHz [29], depending on the chosen daughterboard specification.

This chapter first gives a brief introduction to the USRP in regard to the available resources and FPGA, where-after a description of the hardware structure will me made. Last the firmware of the chosen platform will be analysed with respect to where it is tractable to implement new hardware functionalities.

3.1 Available USRP resources

The USRP is chosen due to the available front ends and the already working SDR[4] FPGA image, which makes it an ideal choice for radio development. However, the choice of this platform also sets some restrictions since it locks the choice of FPGA to the one used in the USRP. Furthermore, the pre-installed FPGA image also uses some of the available fabric. How many resources are available is stated in Table 3.1 for the three available USRPs. The data in this section has been retrieved from the ETTUS homepage.

USRP	USRP1	USRP2	USRPN200
FPGA	Altera Cyclone	Xilinx Spartan XC3S2000	Xilinx Spartan XC3SD1800A
CLB's[5]		5,120 (42 %)	4,160 (54 %)
Logic units		40960 (-)	33280 (-)
RAM (bits)		720,000 (97 %)	1,512,000 (50 %)
Dedicated multipliers		40 (42.5 %)	-
DSP48As	-	-	84 (20 %)
Daughterboard connectors	4	2	2
Firmware upload	USB2	Flash card	Ethernet cable
PC connection	USB2	Ethernet cable	Ethernet cable

Table 3.1: The available USRP's and their resources[30]. The number in parenthesis indicates how much the pre installed firmware uses of the available resources [31]. A dash indicates that the FPGA does not have the given resource.

The advantage of the USRP1 is that it has four antennas, and is able to transmit and receive at different frequencies at different antennas, assuming the prober daugtherboards are installed. However, the USB2 connection is relatively slow (8 Mbps) compared to the USRP2 and N200 which uses a Gigabit Ethernet connection (1000 Mbps). According to Table 2.2 theory Section 2.4,

[4]Software Defined Radio

the sample rate for LTE is between 2 - 30 Msps, rendering the USRP1's communication link too slow. Furthermore the USRP1 has very limited resources, meaning that it is not optimal for this project.

The USRP2 and USRPN200 both have the same firmware but according to table 3.1, the USRPN200 is the most powerful due to the fact that there are more available RAM[6] and that the dedicated multipliers are replaced by the DSP48A slices, which are multipliers targeted towards signal processing functions such as MACC[7] and multiply add [32].

The USRP2 has been chosen as an initial development platform, since it offers a good trade-off between available resources, features and availability. Due to the fact that it uses the same firmware as the USRPN200, it is still possible to transfer the written code to the USRPN200 if there is need for more RAM or multipliers.

3.2 USRP2 hardware structure

As described in the previous section, the USRP2's base is the Spartan-3-2000 FPGA, on which different peripherals has been connected, as shown in Figure 3.1.

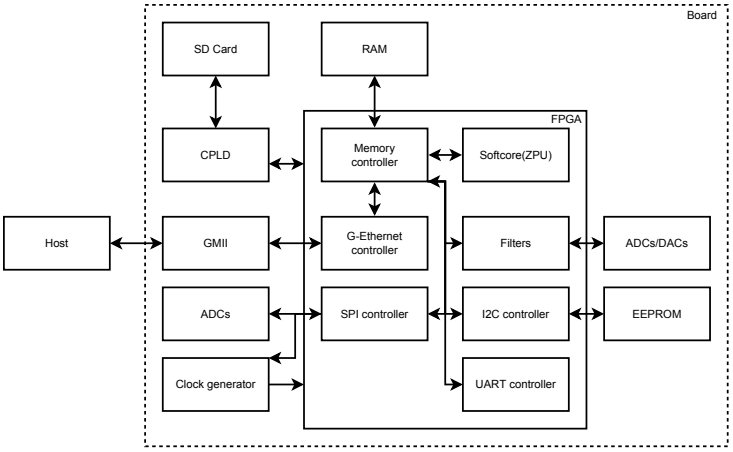


Figure 3.1: Structure of the USRP2 hardware configuration, with the FPGA as the main unit with different connected peripherals[33, p. 14].

The peripherals connected to the USRP2 are used to connect to the daughterboards and the PC, as seen on figure 3.1. The ADC's and DAC's seen on the right-hand side are used to connect to the daughterboards. The GMII[8] peripheral is used to connect to the host PC, whereas the actual communication is controlled by an Ethernet controller implementation using the FPGA fabric.

When turning on the USRP2, the CPLD[9] loads the binary firmware from the SD card onto the FPGA [33]. The CPLD is programmed to handle the

[6]Random Access Memory
[7]Multiply Accumulate, as per Xilinx terminology
[8]Gigabit Media Independent Interface
[9]Complex Programmable Logic Device

initialization process of a standard SD card and when it is initialized it loads the bits from address 0 and onto the FPGA configuration interface, which handles the actual FPGA fabric configuration based on the binary file on the SD card [33]. Once the FPGA image is loaded and initialized, it waits for input from the Host-PC. The communication between USRP2 and Host-PC goes through a Gigabit Ethernet connection.

Since the overall hardware structure of the desired platform has been determined, it is now possible to further analyse the data-flow by taking a closer look on the FPGA image.

3.3 The USRP2's FPGA image

This section describes the interaction of the chosen USRP platform and the Host-PC, with respect to later implementation of Verilog modules which extend the current functionality. Due to lack of documentation, this information has been derived by analysing the firmware designed by Ettus research, which has been supplied such that it is possible for the user to extend the firmware functionality.

The initial analysis is based on a report created by a 8th semester group at Aalborg University [33] and will be referenced to, whereas the rest is deduced from the code itself.

The firmware can be obtained by cloning the UHD[10] repository from *Ettus Research*[11]. When entering the USRP2 firmware path[12] there are different folders, each containing different modules. Table A.1 in appendix A shows the content of the different folders [33].

3.3.1 USRP - Host-PC communication

To implement any extra functionalities on the USRP's FPGA, it is important to understand how it communicates with the Host-PC, such that it is possible to transfer data to and from the FPGA.

The communication between Host-PC and USRP can be incorporated using the UHD library, which is a C++ library that holds different calls, abstracting the user from the lower layers such as the Ethernet protocol and the VRT[13], used to transfer the data [33] between USRP and host.

The UHD library holds different function calls to control the USRP2 transmission, reception and communication. However, since this project has its focus on utilizing the the USRP2 fabric to accelerate some of the LTE physical layer functionalities, the interesting part is how the commands from the Host-PC is interpreted in the USRP2 and how the data from the USRP2 is transferred to

[10] USRP Hardware Driver
[11] git://code.ettus.com/ettus/uhd.git
[12] uhd-repo-path/fpga/usrp2
[13] Vita Radio Protocol, part of the ANSI standard (ANSI/VITA 49.0-2009)

the Host-PC.

The following information has been obtained by reading the supplied Verilog code and tracking the different signals flowing in the code. The code has been obtained as described in Appendix C.

The data from the Host-PC to the USRP2 is streamed through the Ethernet cable using the before mentioned VRT protocol, wrapped in a standard Ethernet protocol. The flow of the Tx stream and Rx stream can be seen in Figure 3.2 and Figure 3.3, respectively. Further information of the routing of data before the VITA deframer can be found in the above mentioned report [33]. The ZPU is a soft-core processing unit, which is implemented on the FPGA and handles setting registers as will be described later in this section.

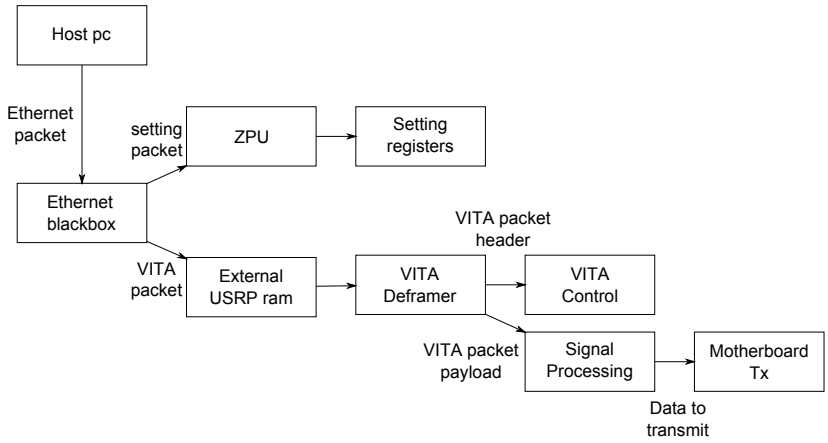


Figure 3.2: The flow of the desired transmitted data from the Host-PC to the Motherboard. Note that Ethernet protocol on top of the VITA packet has been illustrated as a blackbox.

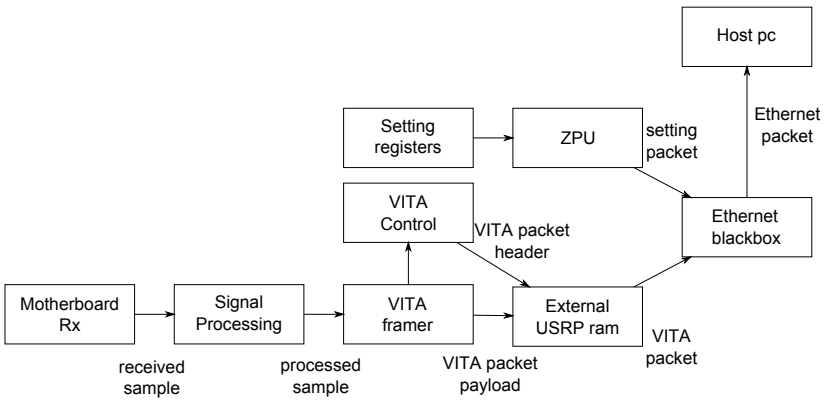


Figure 3.3: The flow of the data received from the USRP2 frontend to the Host-PC. Note that the Ethernet protocol on top of the VITA packet have been illustrated as a blackbox.

There are two main functionalities in the UHD; initialization and data transmission. As seen in Figure 3.2, it is the Ethernet blackbox which checks if the Ethernet packet received from the Host-PC holds setup information or transmission data. Similarly the Ethernet blackbox in Figure 3.3 chooses if the data transmitted to the Host-PC from the USRP2 is information from the setting registers or data from the receiver. The initialization calls are used to setup, or read from, the USRP2's control registers which determines variables as the daughterboard sample rate, choice of filter or buffer size.

If the Ethernet blackbox determines that the received packet controls the USRP2 setup it is routed to the ZPU[14], which then handles the control registers. The control signals are stored in setting registers with different addresses, using a shared bus named *set_data*. Figure 3.4 illustrates the principle of the setting registers usage in the USRP2's FPGA image. The setting register module is name *setting_reg*.

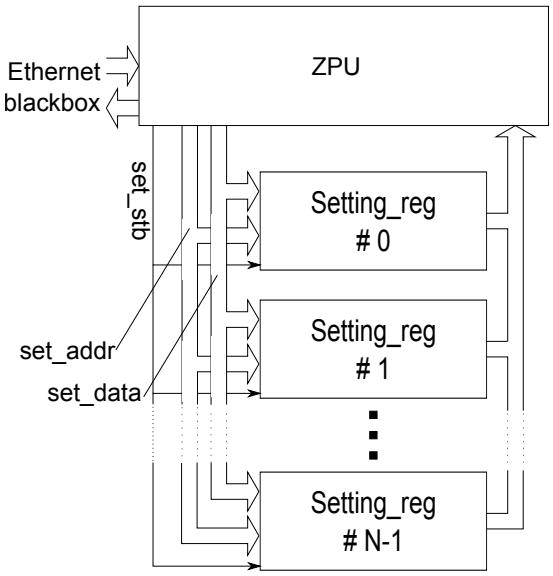


Figure 3.4: A simplified illustration of how the setting registers are connected in the USRP2, deduced from the code itself while doing the initial USRP2 FPGA image analysis described in Appendix C.

When the setting register data strobe *set_stb* goes high, the output of the setting register which has the address defined by the address bus *set_addr* will take the value currently hold by the *set_data* bus.

If the Host-PC asks for information from the setting registers, the ZPU reads the given register and gives the data to the Ethernet blackbox which transmits it to the Host-PC.

The UHD "transmit" and "receive" commands are blocking calls, meaning that it is not possible to setup the USRP2 and transmit at the same time.

[14]Softcore Processing Unit (designed by Xilinx)

When a transmission begins, the USRP2s setting register which controls the transmission is set, and the Ethernet blackbox reroutes the data to the VITA deframer, which ensures that the data is correctly transmitted. Similarly; if a reception is to begin, the setting register controlling this are set and the Ethernet blackbox uses the data input from the DSP core.

This section's focus has been to determine the FPGA resources with easing the implementation process in mind. This has been done by analysing the FPGA firmware structure together with how the C++ library is used to initialize and utilize the USRP.

3.4 FPGA resource analysis summary

The chosen platform has been the USRP2 because of its key features:

- Possibility of using different RF front ends.
- A preprogrammed interface which controls the ADC/DAC interfacing, including initial filters.
- An FPGA with excess fabric for further development.
- High speed USRP to Host-PC communication using a Gigabit Ethernet connection.
- A preprogrammed framework which allows the data to easily get from the FPGA to the Host-PC.

By analysing the dataflow in the USRP it has been concluded, through appendix C, that the best possible place for implementation of additional signal processing would be between the *dsp_core* and *vrt* module as seen in Figure 3.5. This is due to the fact that the *dsp_core* only outputs single samples, using a strobe as control, and that the *vrt* module has some very complex interconnections. By adding the extra signal processing modules in between these blocks, it will be possible for the designer to create a transparent module which is easy to implement.

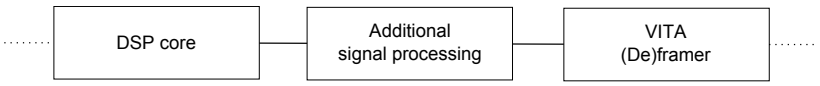


Figure 3.5: Simplified illustration of the best position for implementing the additional signal processing. Note that this figure applies to both the TX and RX chain.

Chapter 4

Implementation Analysis

This chapter will describe the challenges faced when implementing the different physical layer functionalities which have been described through Chapter 2. The initial part of this chapter will focus on describing some of the possible implementation strategies and their advantages and disadvantages. Each section contains a description of the C++ and hardware implementation of a given functionality. The result of this chapter is an analysis of the different functionalities' resource usage on a CPU, based on the C++ program, which together with the implementation analysis will reveal which parts has a possibility of gaining in terms of battery usage or throughput from hardware acceleration. The different physical layer functionalities of this chapter will be analysed and described in the same order they appear in Chapter 2.

4.1 CRC

The CRC attacher is the first part of the transmitter part of the physical layer simulator and its task is to compute and attach the different CRC codes to the packet.

If the CRC is be calculated on the Host-PC using the bitwise polynomial calculation approach, it can take a significant amount of processor power, due to the fact that each modulus 2 addition has to be calculated, and the memory registers has to be updated in value [5]. A way to optimize the calculation would be to implement it by using word-wise divisions and transferring the carry to the next word [5]. This will reduce the calculation time with more than the word length, since the memory register update is also less tedious.

4.1.1 Hardware Implementation - CRC

[5] The CRC encoder is possible to implement in hardware in a very simple, resource effective and fast way by usign the design seen in Figure 4.1.

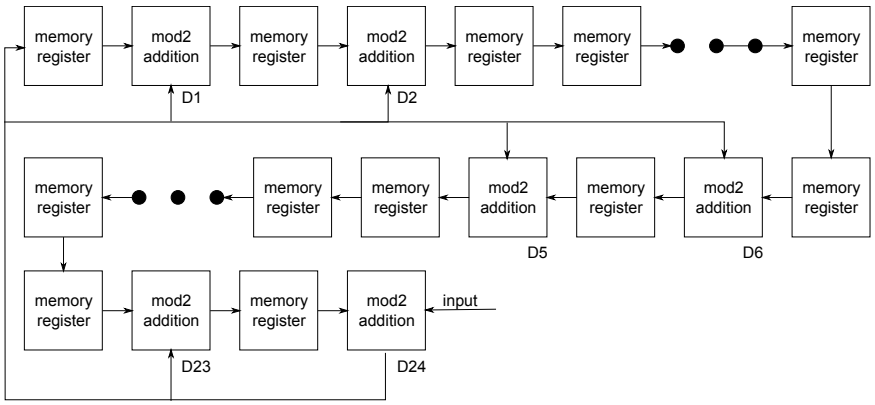


Figure 4.1: Hardware implementation of the CRC_B polynomial of LTE.

Starting all of the memory registers in state zero, the CRC will be present in these when the padding bits have been through. Using the hardware structure depicted in Figure 4.1 it is possible to calculate the CRC at a speed of 1 clock per bit [5]. If this is to be implemented in the FPGA, it would be possible to calculate the CRC fluently while transfering the data from the Host-PC to the PC.

For decoding the memory registers should start in state zero aswell and if these end up in zero after the complete message sequece has been through, the packet is approved.

4.2 Turbo Encoder

The Turbo encoder is responsible for adding the redundant information to the information sequence. As described in Section 2.2, Turbo Codes are encoded by multiple RSC encoders, separated by interleavers.

The RSC encoder in LTE is a rate $1/2$ with the recursive polynomial defined in Equation 4.1 and the forward polynomial as in Equation 4.2, also illustrated in Figure 4.2.

$$g_r = [1011] \tag{4.1}$$
$$g_f = [1101] \tag{4.2}$$

One of the ways to design the RSC encoding process is to make an algorithm which implements the modulus two additions (**XOR**).

It is clear from Equation 4.1 that the Turbo encoder uses roughly four bitwise **XOR** additions and four separate memory operations. Assuming that these are all simple operations for a CPU thus only require only one clockcycle per operation, this design costs approximately 7 clockcycles per bit.

Another design of the RSC encoder can be realized by taking advantage of the trellis structure as seen in Section 2.2.1. The FSM on Figure 2.7 is a Mealy FSM since the output is based on the current encoder state and if the new

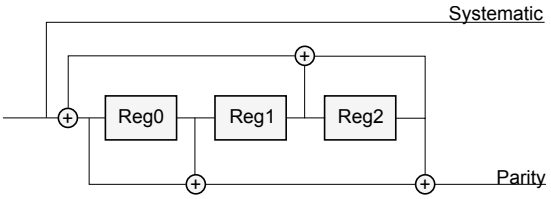


Figure 4.2: The RSC encoder used in the LTE turbo encoder. As illustrated, it consists of three one-bit memory registers, connected with **XOR** additions.

```
1  for (i = 0; i => framelength; i++) {
2    if(i < framelength - 3) {
3      u[i] = input[i];
4      feedback = input[i] ^ delayvec[1] ^ delayvec[2];
5      z[i] = feedback ^ delayvec[0] ^ delayvec[2];
6      delayvec[2] = delayvec[1];
7      delayvec[1] = delayvec[0];
8      delayvec[0] = feedback;
9    } else {
10     feedback = u[i] ^ delayvec[1] ^ delayvec[2];
11     u[i] = feedback;
12     z[i] = delayvec[0] ^ delayvec[2];
13     delayvec[2] = delayvec[1];
14     delayvec[1] = delayvec[0];
15     delayvec[0] = 0;
16   }
17 }
```

Listing 4.1: C++ code which implements the Turbo encoder using additions. **input** are the input sequence to the encoder **u** are the systematic output **z** are the parity bit and **delayvec** are the memory registers.

```
1  trellis.fromstates[current state][input bit][desired output]
```

Listing 4.2: Definition of the FSM array for the turbo encoder. *current state* is the current state of the receiver (the previous *next state*) and *input bit* is the current information bit to be encoded and *desired output* holds either: 0:next state and 1:systematic bit and 2:parity bit.

input is 1 or 0. This way of presenting the encoding process can be used in the implementation process by creating a lookup table which holds the different transitions given a certain state and input.

$$\begin{aligned} mem_{\text{lookup table}} &= N_{\text{states}} \cdot N_{\text{input}} \cdot N_{\text{desired output}} \\ &= 2^{\text{memory registers}} \cdot 2 \cdot 3 = 96 \qquad [kB] \end{aligned} \tag{4.3}$$

Using a three dimensional array as in Listing 4.2 increases the memory usage by 96 kB, compared to the previous design, as shown in Equation 4.3. However; the encoding process can be done with roughly three memory operations as seen in Listing 4.3. Making the same assumption as before, this design will have a clockcycle per bit use of three, meaning it will be more than twice

```
1  for (i = 0; i => framelength; i++) {
2    if(i < framelength-3) {
3      current_state = trellis.fromstates[current_state][input[i]][0];
4      u[i] = input[i];
5      z[i] = trellis.fromstates[current_state][input[i]][2];
6    } else {
7      if (NextStates[CurrentState][1][0] > NextStates[CurrentState][0][0]){
8        current_state = NextStates[CurrentState][1][0];
9        u[i] = NextStates[CurrentState][1][1];
10       z[i] = NextStates[CurrentState][1][2];
11     } else {
12       current_state = NextStates[CurrentState][0][0];
13       u[i] = NextStates[CurrentState][0][1];
14       z[i] = NextStates[CurrentState][0][2];
15     }
16   }
17 }
```

Listing 4.3: How the lookup table is used to encode the Turbo codes

```
1   real(symbol[k]) = u[k];
2   imag(symbol[k]) = u_int[k];
```

Listing 4.4: Example of a C++ implementation where both the systematic and systematic interleaved bit are transmitted using only one sample transmission on the USRP2.

as fast as the previous design.

For implementation on the CPU it has been chosen to use a lookup table approach due to increased speed and the fact that there are vast amounts of available memory on the Host-PC. In the C++ program, the creation of the FSM lookup table is done in a separate class, *polytotrellis*, since it is also used in the Turbo Decoder. The *polytotrellis* class generates the FSM lookup table by exposing the RSC code polynomial through all possible current states with all possible inputs, and storing the inputs in the three dimensional array *transitions*.

4.2.1 Hardware Implementation - Turbo Encoder

Concerning hardware implementation of the Turbo encoder, the RSC encoder uses a very small amount of resources [11]. However, the interleaver polynomial will need some design thoughts regarding resource usage [11], since it uses two multiplications and one division if the algorithm were to be implemented, or a lot of memory for a lookup table.

One way to avoid implementing the interleaver on the FPGA, would be to interleave the data on the Host-PC and then transmit both data and interleaved data in the same packet. This can easily be done by utilising the fact that the Host-PC and USRP2 communication is done by a $2 \cdot 16$ bit number, which can be used to transmit the binary sequence, as seen in Equation 4.4.

Furthermore the above trick can also be used to transmit multiple bits in one sample, decreasing the load on the Ethernet connection by $\frac{1}{\mathrm{N}_{bits/\mathrm{sample}}}$. Note that sample does no longer hold a complex value, since it holds bits to be converted into a message. The upper limit of number of bits which can

be transmitted per packet is 16, due to the fact that this is the size of the sample. However, the Turbo encoder's speed is limited by the FPGA master clock, since the encoder cannot operate faster than one master clock cycle [11]. Hence; the maximum number of bits (if below 16) which can be transmitted are:

$$N_{bits/\text{package}} = \frac{100000000}{\text{sample rate}}$$

4.3 Rate Matching

The rate matching works by two different interleavers, as described in Section 2.2.2. However, a closer inspection of the described interleaver for b^0 and b^1 shows that it is the same as for b^2, just offset by -1.

$$\Pi_{b^0}(k) = \Pi_{b^1}(k) = \left(P\left(\left\lfloor \frac{k}{R} \right\rfloor\right) + 32\,(k \bmod R)\right) \bmod N_{array} \tag{4.4}$$

$$\Pi_{b^2}(k) = \left(P\left(\left\lfloor \frac{k}{R} \right\rfloor\right) + 32\,(k \bmod R) + 1\right) \bmod N_{array} \tag{4.5}$$

The C++ implementation of the (de)interleaver has been done by using a lookup table for the permutation order P, seen in Equation 2.6, and then using the two defined mathematical expressions to calculate the address from which the current bit should be read from.

4.3.1 Hardware Implementation - Rate Matching

The Equations used to implement the rate matching are not preferable to implement on a hardware platform, due to the fact that is uses two multiplications, one division and two modulus functions. It does have the advantage that the array P is equal to a bit reversed 5 bit number and that the multiplication of 32 can be done by shifting 5 times to the left. However, it has been researched if there is a more convenient way of implementation.

The research was initially started due to the fact that the actual output of the interleaver polynomial resembled a normal bit reversing. Because of this, the output of the rate matching algorithm of b^0 and b^1 was held up against the output of the bit reversing, a clipping of this is seen in table 4.1.

$$\Pi_{b^0}(k) = \begin{cases} \Pi_{b^0}(k-1) + 32 & \text{for } (k \bmod 8) \neq 0 \\ bitrevorder(k) & \text{otherwise} \end{cases} \tag{4.6}$$

Through the analysis it was clear that the rate matching algorithm can also be described by Equation 4.6, which is implementable by a cost effective structure depicted on Figure 4.3.

index (k)	Bit reversing	Rate matching	Rate matching $\Pi_b^0(k) - \Pi_b^0(k-1)$
0	0	0	0
1	128	32	32
2	64	64	32
:	:	:	:
8	16	16	16
9	48	144	32
:	:	:	:
16	8	8	8
17	40	136	32
:	:	:	:
255	255	255	32

Table 4.1: Clipping of the output from a MATLAB analysis of the function *bitrevorder* and the rate matching algorithm for b^0 and b^1. The $\Pi_b^0(k) - \Pi_b^0(k-1)$ illustrated that the previous calculated index of the rate matcher are subtracted from the current.

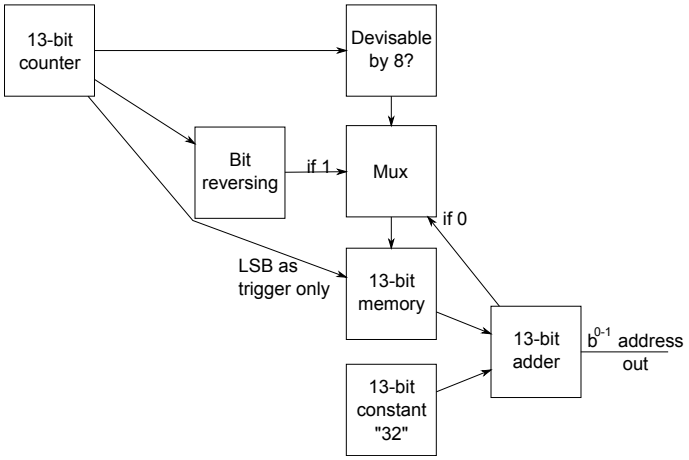


Figure 4.3: The hardware implementation of the rate matching index calculator.

The implementation of the rate match index calculator in hardware can be rather low cost since:

- The counter only consists of enough memory registers, such that it is able to count to the maximum address (in case of LTE this is $\lfloor \log 2(6044) \rfloor =$ 13 bit) and an adder.
- Bit reversing can be done purely by routing cost
- The mux which chooses which address to use is a 13 bit input to 13 bit output mux with only two possible addresses.
- The adder is 13 bit which cannot overflow
- Only 13 registers are used to store the number 32.

```
1   de-scramble the rate matched input using the reverse function
2
3   initialize extrinsic1 and extrinsic2 to 0
4
5   calculate gamma for encoder 1
6   calculate gamma for encoder 2
7   for up to the desired max iterations do:
8     calculate the soft bits output for RSC decoder 1
9     interleave the soft bits of decoder 1 and make them extrinsic
10
11    calculate the soft bits output for RSC decoder 2
12    check if the CRC checks out on the hard output of RSC decoder, if it does
          then stop decoding
13
14    interleave the soft bits of decoder 2 and make them extrinsic
15  }
```

Listing 4.5: The pseudo code for the chosen Turbo decoder implementation.

The memory registers which holds the input and the corresponding output mux can however be costly, but this is hard to avoid since it is required to store the whole sequence due to the interleaving.

To implement the index calculator for b^2 an extra adder which adds 1 bit should be used.

It has been shown that the rate matching algorithm is possible to implement rather cost effectively on a FPGA hardware platform through a analysis of the output values.

4.4 Turbo Decoder

The Turbo decoder is, as described in Section 2.2, created by serial connected RSC decoders which exchanges soft valued extrinsic information about the decoded bits to iteratively increase the decoding performance. The interleaver and deinterleaver of the decoder are similar to the ones in the encoder, hence the focus in this section will be implementational issues of the RSC decoder.

To increase the speed of the Turbo decoder, it has been taken advantage of the fact that the transition possibilities γ only has to be calculated once, since these are based on the received bits and therefore does not change through the iterative process. This leads to the pseudo-code of implementation as seen in listing 4.5.

Table 4.2 [4] shows the complexity of the possible RSC decoder algorithms and is used to help decide which algorithm should be used for implementation.

For the CPU implementation, the BCJR algorithm is chosen because of the fact that this is the one which will be analysed with respect to hardware implementational analysis.

4.4.1 Hardware Implementation - Turbo Decoder

The hardware implementational aspect of the Turbo decoder has been exhaustively researched, since it is a very calculation heavy algorithm, which can gain from hardware acceleration. Firstly in this section it is analysed if it is in

	BCJR (MAP)	Max-Log-MAP	Viterbi	SOVA
add	$2 \cdot 2^k \cdot 2^v + 6$	$4 \cdot 2^k \cdot 2^v + 8$	$2 \cdot 2^k \cdot 2^v + 2 \cdot 2^v$	$2 \cdot 2^k \cdot 2^v + 9$
multi	$5 \cdot 2^k \cdot 2^v + 8$	$2 \cdot 2^k \cdot 2^v$	$2^k \cdot 2^v$	$2^k \cdot 2^v$
Max. ops		$4 \cdot 2^k \cdot 2^v$	2^v	$2 \cdot 2^v - 1$
exp	$2 \cdot 2^k \cdot 2^v$			

Table 4.2: Comparison of the four RSC decoder algorithms calculation complexity for a single time sample relative to each other[4, p.153]. Note that all the algorithms apart from the Viterbi algorithm has also backwards recursive part, hence the SOVA, BCJR and MAX-Log-MAP algorithms uses double the amount of time steps as the Viterbi. k is the number of information input bits to the RSC encoder and v is the constraint length of the encoder.

any way plausible to obtain the desired throughput of the Turbo decoder and thereafter an analysis of how plausible it is to implement this on the USRP2 firmware.

Recent studies [34], [35] have focused on the hardware implementation of the Max-Log-MAP algorithm, due to its low complexity compared to the BCJR. The advantage of the Max-Log-MAP is that it only works on the best possible transition, and hereby greatly reducing the energy per processed bit E_b^{pr} and enables high throughput. However, the reduced complexity have a cost of 0.5 dB [36] BER, which in the end leads to an increase in needed transmitter energy E_b^{tx} of 10 % [37]. If the nodes are separated by dozens of meters or more, the overall energy consumption $E_b^{pr} + E_b^{tx}$ when using the Max-Log-Map surpasses the energy consumption by using the more complex BCJR [37].

To analyse whether it is plausible to obtain throughput compliant with the LTE standard, best case scenario optimization possibilities has been found. This means that these are theoretical optimizations which initially does not take resource usage into account. The reason for making this analysis is that this will quickly show whether it is worth to continue with the analysis.

The BCJR algorithm is the logic choice for an initial optimization analysis, due to it's high complexity. In the BCJR algorithm, the bit to be currently calculated is dependant on the result of the value of the previous calculation, hence it is not possible to calculate multiple adjacent bits in parallel. It is however possible to calculate each current transition probability, seen in the trellis diagram Figure 2.7 in parallel.

Utilizing the parallelization of the transition calculations, speed can be increased by up to

$$\begin{aligned} r_{\text{transition}} &= 2^{N_{\text{constraint}}} \cdot N_{transitions/\text{state}} && (4.7) \\ &= 2^4 \cdot 2 = 32 && [-] \end{aligned}$$

where:

- $r_{\text{transition}}$ is the rate of improvement of the transition calculation speed when comparing to a single core processor.

- $N_{\text{constraint}}$ are the constraint length of thee RSC encoder.
- $N_{transitions/\text{state}}$ are the number of transitions per state.

The backward polynomial β can be calculated parallel with the forward polynomial α, which leads to a rate increase of the α and β total calculation time of $r_{\alpha \text{ and}\beta} = 2$.

An optimization using lookup table values, as in [37], can theoretically make both the α, β and γ calculation take only one clock cycle. With parallel β and α calculations, the LLR calculation can begin when these metrics has been halfway calculated, e.g. when the index k has reached the value of $N_{bits}/2$, since the LLR calculation is dependant on α_{k-1}, γ_k and β_k. Using the currently described optimization methods, the number of clock cycles for a Turbo decoding algorithm would be as in Equation 4.9.

$$N_{clk/\text{BCJR}} = N_{\text{bits}} + \frac{N_{\text{bits}}}{2} \tag{4.8}$$

$$N_{clk/\text{Turbo decoding}} = 2N_{\text{iterations}} \cdot N_{clk/\text{BCJR}} \tag{4.9}$$

Where:

- $N_{clk/\text{BCJR}}$ is the total number of clocks for 1 full BCJR decoding of the received bit sequence
- N_{bits} is the number of bits in the output sequence $\tilde{m}$
- $N_{\text{iterations}}$ is the maksimum number of iterations the Turbo decoder can run
- $N_{clk/\text{Turbo decoding}}$ is the clocks which is used for a full turbo decoding run of the input sequence

Using Equation 4.9, a packet length of 6044 as defined in LTE, the 100 MHz clock on the USRP2 and 5 iterations the maximum throughput of the current optimizations will be equal to:

$$\begin{aligned} throughput &= N_{clk/\text{bit}} \cdot f_{\text{FPGA}} \quad \text{[bps]} \\ &= \frac{N_{\text{bits}}}{N_{clk/\text{Turbo decoding}}} \cdot f_{\text{FPGA}} \\ &= 5\frac{6044}{2\left(6044 + {}^{6044}/_{2}\right)} \cdot 100000000 \\ &= 6666666.\overline{66} \end{aligned} \tag{4.10}$$

The result seen in Equation 4.11 is not sufficient when dealing with LTE's high possible throughput of up to 200 Mbps. However, the high LTE throughput assumes a 4x4 MIMO system and 20 MHz bandwidth. This report does only focus on SISO and the current bandwidth are only 10 MHz, which leads to the output being only $\frac{200}{4 \cdot 2} = 33$ Mbps.

It is possible to further increase the speed of the Turbo decoder e.g. by insertion of pipelines, and with the fact that the system does only need a rate increase of 5, as seen in Equation 4.12, it seems little plausible to implement the Turbo Decoder on the USRP2, regarding throughput.

$$r_{\text{Further optimizations}} = \frac{throughput_{\text{desired}}}{throughput_{\text{current}}} \tag{4.11}$$

$$= \frac{33333333}{6666666} \approx 5 \tag{4.12}$$

Publication	[37]	[38]	[39]	[40]	[34]	[35]
Algorithm	LUT[1]-Log	LUT-Log	LUT-Log	LUT-Log	Max-Log	Max-Log
Block size (bit)	6144	5114	5114	5114	6144	6144
Technology (nm)	90	180	180	180	65	120
Supply voltage (V)	1.0	1.8	1.8	1.8	-	1.2
Area A (mm2) (Scaled for 90 nm)	0.35	9 (2.25)	14.5 (3.63)	8.2 (2.05)	2.1 (4.0)	3.57 (2.0)
Gate count (exclusive of memory)	7.5k	85k	410k	65k	-	553k
Memory required (kbit)	188	239	450	161	-	129
Clock frequency F (MHz)	333	111	145	100	300	390.6
Decoding iterations	5	10	8	6.5	6	5.5
Throughput T (Mb/s)	1.03	2	10.8	4.17	150	390.6
Power consumption (mW) (Scaled for 90 nm)	4.17	292 (36.5)	956 (119.4)	320 (40)	300 (796.4)	788.9 (332.8)
Energy consumption (nJ/bit/iteration) (Scaled for 90 nm)	0.4	14.6 (1.8)	11.1 (1.4)	12.7 (1.59)	0.31 (0.81)	0.37 (0.16)
$E_b^{tx} + E_b^{pr}$ (nJ/bit) when transmitting over 39 m (5 iterations)	10.16	17.16	15.16	16.06	13.42	10.17
$E_b^{tx} + E_b^{pr}$ (nJ/bit) when transmitting over 58 m (5 iterations)	41.92	48.92	46.92	47.82	49.88	46.63

Table 4.3: Comparison of resource usage and power consumption of already proposed architectures of the Turbo Decoder [37].

Equation 4.12 shows that it seems difficult to implement the Turbo decoder on the USRP2. Furthermore Table 4.3 shows that the currently suggested high throughput performance Turbo decoders uses up to 553000 logic gates, which well exceeds the available 40960 on the Spartan 3 FPGA available, described in Table 3.1. The high resource cost of the other proposed architectures,implies that it is not a tractable solution to implement the Turbo decoding on the USRP2. However, it could be possible by using the high speed FPGA extension port of the USRP2, which would also enable a FPGA that uses a higher clock frequency to be used. A higher clock frequency will also further reduce the extra needed optimizations to Turbo decode.

4.5 Bit-(de)modulation

The bit-modulation is the functionality of transferring the binary message onto symbol which are to be mapped onto subcarriers in OFDM. The bit demodulation interprets the symbol on each received subcarrier and decodes them into

one, or multiple, soft bits.

Due to the fact that there are plenty of memory available on the Host-PC, it was chosen that the modulation of the bits should be done by the use of a lookup table. This is realised by making a two dimensional array which is defined as in Listing 4.6.

```
1   ***Bit_mod_array = [modulation form][binary value]
```

Listing 4.6: The definition of the array which holds the lookup tables for the bit-modulation.

Where

- Modulation form ranges from 0 to 3 and 0=bpsk, 1=QAM, 2=16-QAM, 3=64-QAM.
- Binary value are the sum of the binary bits e.g. bpsk has the value 0 for the bit value 0 and 1 for the bit value 0, whereas QAM has 0 for the bits 00, 1 for 01, 2 for 10 and 3 for 11.

The decoding of the received symbols into bits are done by a straight forward implementation of the algorithms seen in Equaions 2.21 to 2.29. This result in a relatively optimized decoding process, using only a very few amount of operations per bit.

4.5.1 Hardware Implementation - Bit Modulation

On the Host-PC, the bit-mapping is done using a lookup table, which is chosen due to the increasing complexity of writing an algorithm, compared to the memory usage of a lookup table. There are many different approaches to design a hardware implementation of this functionality. The focus in this report will be the difference of a pure lookup table design versus a algorithm design.

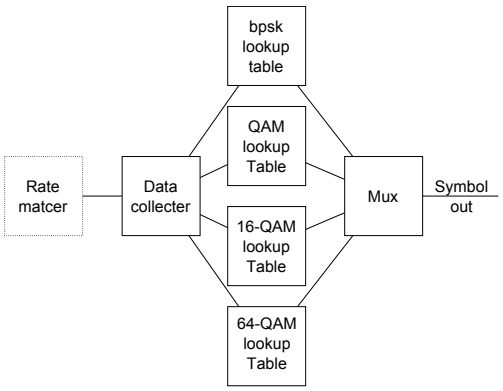
In Figure 4.4, the *data collector* is a multi access bit-wise memory storage of the input from the Turbo encoder, which works as a serial to parallel converter. The inputs are written into the 6-bit output register of the *data collector* as seen in Equation 4.13 to 4.16. During the next coming resource usage estimation, the *data collector* is seen as having equal resource usage for both implementations.

$$dc_{PSK}(k) = [m(k),\ 0_{4:0}] \tag{4.13}$$
$$dc_{QAM}(k) = [m(2k:2k+1),\ 0_{3:0}] \tag{4.14}$$
$$dc_{16-QAM}(k) = [m(4k:4k+3),\ 0_{1:0}] \tag{4.15}$$
$$dc_{64-QAM}(k) = [m(6k:6k+5)] \tag{4.16}$$

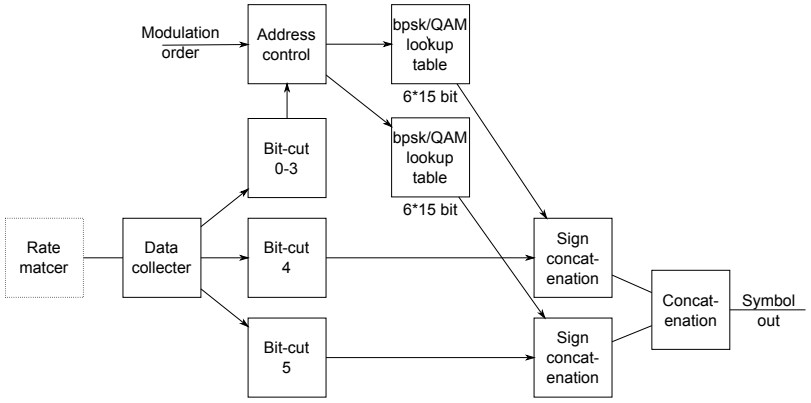
Where:

- dc_{suffix} is the resulting contents of the 6-bit wide memory register for the given modulation order, defined by $suffix$
- Square brackets indicate a concatenation from MSB [2] to LSB [3]; [MSB, ..., LSB]

[2] Most Significant Bit
[3] Least Significant Bit



(a) Basics of the lookup table implementation.



(b) Block diagram of the algorithm implementation.

Figure 4.4: Two different block diagrams which illustrates the basics of the bit-mapping hardware implementation.

- $0_{x:0}$ indicates that zeroes are filled in the registers from x to 0

The resource usage for the lookup table implementation seen in Figure 4.4a have been estimated in two steps. Firstly the memory use has been estimated to correspond to the total number of symbols in all of the modulation schemes:

$$N_{\text{mem regs}} = 32(64 + 16 + 4 + 2) \quad [\text{bits}]$$

The *Mux* in Figure 4.4a is a 32 output mux with 32 inputs for each symbol options, resulting in the mux to be a 4*32 input to 32 output mux. If the lookup table is implemented using onboard RAM, this are the only mux resource usage, however, if the lookup table are implemented on the fabric, an address mux will also use resources.

The implementation of the algorithm, seen in Figure 4.4b takes advantage of the fact that the input bits to the symbol mapper, controls different parts of the symbol generation, as seen in table 4.4.

The implementation shown in Figure 4.4b also uses a lookup table for each symbol. However, in this case it only contains values of the possible in-phase

Input bit nr (0 is LSB)	Controls
BPSK	
0	Sign of imag and real value.
QAM	
0	Sign of imag value.
1	Sign of real value.
16-QAM	
0-1	Real and imag value.
2	Sign of imag value.
3	Sign of real value.
64-QAM	
0-3	Real and imag value.
4	Sign of imag value.
5	Sign of real value.

Table 4.4: illustration of how the input bits to the bit-mapper controls different parts of the encoding process, due to the nature of the mapping scheme defined in LTE.

or quadrature bit coordinates for the different constellations which are defined in LTE as:

$$\text{BPSK and QAM} = {}^{1}/\sqrt{2} \quad [-] \quad (4.17)$$

$$16 - \text{QAM} = ({}^{1}/\sqrt{10},\ {}^{3}/\sqrt{10}) \quad [-] \quad (4.18)$$

$$64 - \text{QAM} = ({}^{1}/\sqrt{42},\ {}^{3}/\sqrt{42},\ {}^{5}/\sqrt{42}) \quad [-] \quad (4.19)$$

BPSK and QAM can reuse the same values and the system does add a sign bit later in the process, hence the memory usage are equal to:

$$N_{\text{mem regs}} = 15(1 + 2 + 3) \quad \text{[bits]}$$

The address control to the lookup tables are handled by a logic circuit, which calculates which input should be available using two inputs; the modulation order, using a 2 bit input and the desired value of the bit, described by up to four bits. The circuit follows the truth table seen in Table 4.6. Note that the input to the logic circuit is from the *data collector* which has output as in Equation 4.13 to 4.16.

When concerning the two implementation options the lookup table uses up more of the RAM (or onboard registers) than the algorithm implementation due to the fact that it has to contain all possible symbol values. The algorithm uses less memory than the lookup table, however the amount of LUTs are higher since these are used to implement the logic which controls the address of the symbol coordinates according to Section 2.13. The speed of the two circuits are equal and the designer is able to clock out symbols at a rate of 6 times the symbol rate.

Based on the initial implementation analysis, it is deduced that the choice of implementation will rely on the resources available for implementation on

Input						Output			Output		
Mod order		Address control				real addr			imag addr		
0	0	0	0	0	0	0	0	0	0	0	0
		-					-			-	
0	1	0	0	0	0	0	0	0	0	0	0
		-					-			-	
1	0	0	0	0	0	0	0	1	0	0	1
1	0	0	1	0	0	0	0	1	0	1	0
1	0	1	0	0	0	0	1	0	0	0	1
1	0	1	1	0	0	0	1	0	0	1	0
		-					-			-	
1	1	0	0	0	0	1	0	0	1	0	0
1	1	0	0	0	1	1	0	0	0	1	1
1	1	0	0	1	0	0	1	1	1	0	0
1	1	0	0	1	1	0	1	1	0	1	1
1	1	0	1	0	0	1	0	0	1	0	1
1	1	0	1	0	1	1	0	0	1	1	0
1	1	0	1	1	0	0	1	1	1	0	1
1	1	0	1	1	1	0	1	1	1	1	0
1	1	1	0	0	0	1	0	1	1	0	0
1	1	1	0	0	1	1	0	1	0	1	1
1	1	1	0	1	0	1	1	0	1	0	0
1	1	1	0	1	1	1	1	0	0	1	1
1	1	1	1	0	0	1	0	1	1	0	1
1	1	1	1	0	1	1	0	1	1	1	0
1	1	1	1	1	0	1	1	0	1	0	1
1	1	1	1	1	1	1	1	0	0	1	1

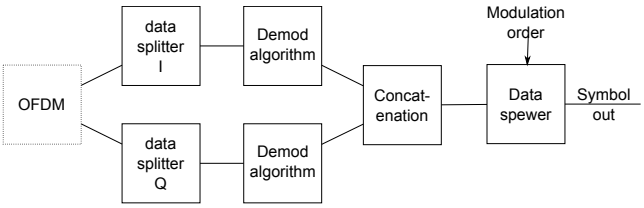
Table 4.5: Truth table for the circuit which controls the address input to the mux in the implementation seen on Figure 4.4b, note that the transitions not listed are impossible states.

the FPGA. There are no immediate optimal solution when it comes to speed or fabric usage, since both choices uses around the same amount, however the resources being different from each other.

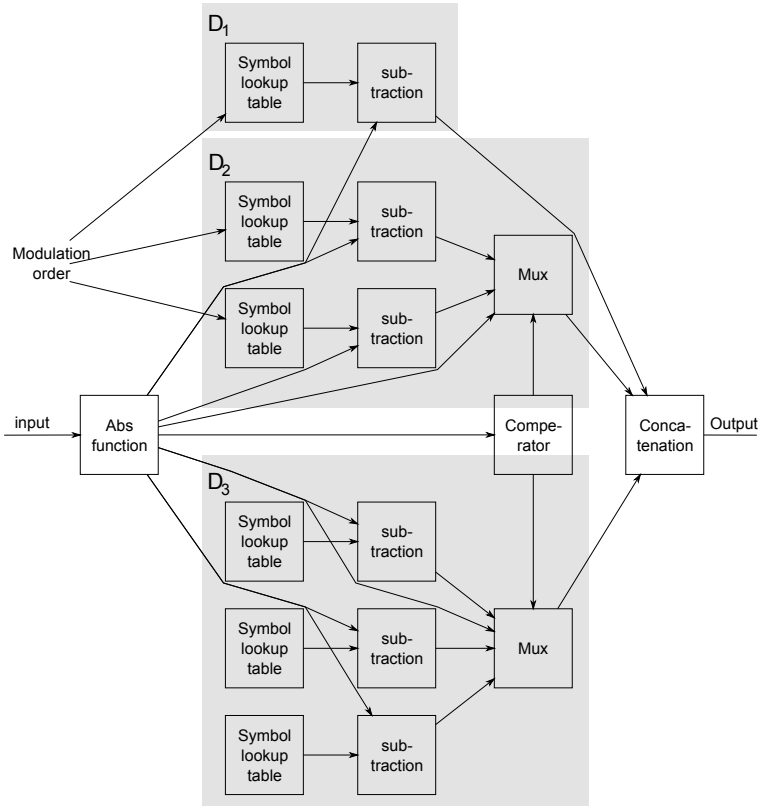
4.5.2 Hardware Implementation - Bit Demodulation

The de-mapping happens by using the simplified LLR calculation seen in Equation 2.20 in the algorithms seen in Section 2.4.1.1 to 2.4.1.4. Due to the simplification the soft output demapper only needs; an "absolute value" function, a subtraction, the available values of the symbol and some data routing/multiplexing.

An option to consider regarding the implementation of the soft-output demapper is whether the decoder should decode all the bits in parallel or serial. If the symbols are to be decoded in parallel, the bit rate is equal to the symbol rate times the number of bits per symbol.



(a) Overall bit demapper structure.



(b) Close-up on the *Demod algorithm* block used in the overall bit demapper structure.

Figure 4.5: The implementational suggestion for the soft-output symbol demapper.

The *data spewer* seen in Figure 4.5 is a buffer which ensures that the output bits can be read at any given rate desired up to the symbol rate times the bits per symbol.

4.5.3 OFDM

As described in Section 2.4.2 the OFDM symbol is (de)modulated using an (I)FFT, due to the fact that it is an operation wise optimized version of the DFT. The DFT can transfer the subcarriers between time and frequency domain with very precise subcarrier spacing. There are different algorithms which

can be used to implement the IFFT whereas one of the most exhaustively researched is the Cooley Tukey algorithm, which will briefly be described here.

4.5.3.1 Cooley Tukey

The Cooley Tukey FFT algorithm is a mathematically rearranged version of the DFT, that utilizes that the DFT can be split up into two sums, consisting of the DFT of the even indexes and the DFT of the odd indexes as is seen in Equation 4.20

$$\begin{aligned} \mathrm{DFT}\left(X\right) &= \mathrm{DFT}\left(X_{even}\right) + \mathrm{DFT}\left(X_{odd}\right) \\ \sum_{n=0}^{N-1} x_n \exp\left(-\frac{2\pi i}{N} nk\right) &= \sum_{m=0}^{N/2-1} x_{2m} \exp\left(-\frac{2\pi i}{N}\left(2m\right)k\right) \\ &\quad + \exp\left(-\frac{2\pi i}{N}k\right) \sum_{m=0}^{N/2-1} x_{2m+1} \exp\left(-\frac{2\pi i}{N}\left(2m\right)k\right) \end{aligned} \tag{4.20}$$

This split can then be repeated until the DFT calculation is only a radix-2 butterfly calculation.

The main function in the Cooley Tukey algorithm is the butterfly operation, which is roughly seen a 2 point DFT, as illustrated in Figure 4.6.

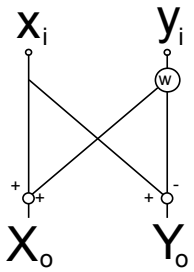


Figure 4.6: Illustration of a single butterfly operation, where x is equal to input 0, y is equal to input 1 and w is the twiddle factor. As seen on the Figure it is possible to make the calculations in-place, e.g. storing the data at the memory it came from.

In Figure 4.6 w indicates the use of the twiddle factor, which is equal to:

$$w = \exp -j\pi^{n}/_{N} \tag{4.21}$$

Where:

- n is the number of sub butterflies used (as illustrated in the nextcoming Figure 4.7
- N is the number of sub butterflies present in the current stage.

The butterfly operation in Figure 4.6 are calculations using complex numbers as seen in Equation 4.22 and 4.23.

$$(X_r + X_i) = (x_r + x_i) + (w_r + w_i) \cdot (y_r + y_i) \tag{4.22}$$
$$(Y_r + Y_i) = (x_r + x_i) - (w_r + w_i) \cdot (y_r + y_i) \tag{4.23}$$

Complex operations are not a base function of the FPGA/C++ language, therefore it it will ease the implementation if the different calculations would be deducted. By rearranging Equation 4.22 and 4.23 it is possible to gain four expressions as seen below in Equation 4.24 to 4.27.

$$X_r = x_r + w_r \cdot y_r - w_i \cdot y_i \tag{4.24}$$
$$X_i = x_i + w_r \cdot y_i + w_i \cdot y_r \tag{4.25}$$
$$Y_r = x_r - w_r \cdot y_r + w_i \cdot y_i \tag{4.26}$$
$$Y_i = x_i - w_r \cdot y_i - w_i \cdot y_r \tag{4.27}$$

By combining more of these butterfly operations, it is possible to scale the FFT size in the range of 2^n with n being an integer. Figure 4.7 illustrates an 8-point FFT.

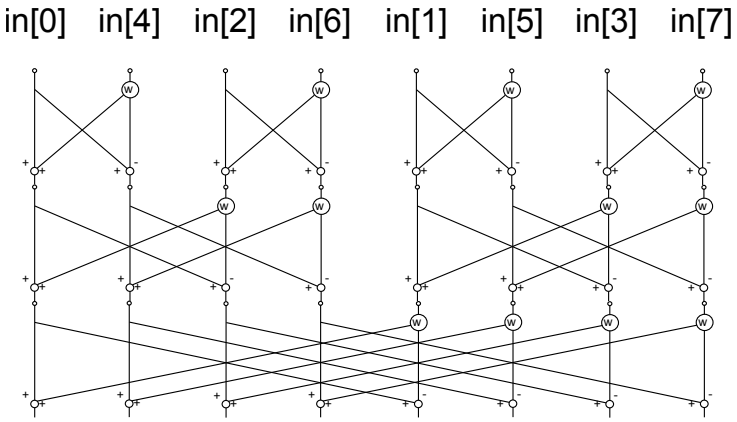


Figure 4.7: An 8-point FFT using the Cooley Turkey algorithm. Note that the inputs to the FFT should be bit-reversed in order for the algorithm to work.

Note that the input addresses has been bit reversed, which was done due to the fact that the algorithm takes advantage of the cyclic properties of the FFT, as seen in Equation 4.20.

The IFFT are calculated by using the following steps, which includes the FFT:

1. Interchange the imaginary and real parts
2. Calculate FFT
3. Scale by $\frac{1}{N_{\mathrm{FFT}}}$
4. Interchange the imaginary and real parts

```
1  // code to be placed outside the actual fft loop
2    if (N_fft = 128)
3      n_offset = 16;
4    else if (N_fft = 256)
5      n_offset = 8;
6    else if (N_fft = 512)
7      n_offset = 4;
8    else if (N_fft = 1024)
9      n_offset = 2;
10   else
11     n_offset = 0;
12
13 // code which accesses the BaseBandSymbol array, given that the counter "k"
       are used as a non bit reversed FFT input address
14   BaseBandSymbol[bitrevorder[n + n_offset]];
```

Listing 4.7: How to enter the *BaseBandSymbol* (of length 2048 + CP length) array using the bit-reversed addresses in the *bitrevorder* (of length 2048) array.

4.5.3.2 CPU Implementation

The implementation on the CPU uses the radix-2 butterfly structure to calculate the (I)FFT. The input address bit reversion is handled by a lookup table, due to the fact that one single table can hold the values for all of the fft lengths, if given the right offset when read from, due to the nature of bit reversing. Given the array *BaseBandSymbol* which holds the OFDM symbol and the array *bitrevorder*, which holds the bit reversing lookup table, listing 4.7 shows an example of how to enter the *BaseBandSymbol* array in the C++ code.

The implementation of the radix 2 algorithm are done by following Equation 4.24 to 4.27 and then replacing xr, xi, yr, yi by the array *BaseBandSymbol* accessed using the correct indexes.

The access control of the FFT is controlled by three loops; one loop to determine which stage the calculation is in, one which determines which pair the system operates on and at last, one which determines which radix-2 algorithm in the specific pair is calculated.

From the 8-point FFT illustration in Figure 4.7 it is deduced that:

$$N_{\text{stages}} = \log 2\,(N_{\text{FFT}}) \tag{4.28}$$

$$N_{\text{butterflies}} = \frac{N_{\text{FFT}}}{2} \tag{4.29}$$

$$n = 2 \cdot n_{butterfly} + n_{pack} \cdot 2^{n_{stage}} + x0\_or\_y1 \tag{4.30}$$

Where:

- N_{stages} is the number of stages
- $N_{\text{butterflies}}$ is the number of radix-2 operations per stage.
- n is the non-bitreversed index of the variable *BaseBandSymbol*
- n_{pack} holds the current pair number of the current state
- $n_{butterfly}$ indicates which butterfly in the current pack the FFT uses
- n_{stage} is the current state of the calculation

- $x0\_or\_y1$ is a variable determining whether the x(variable is 0) or y(variable is 1) input to the butterfly calculation is to be used.

Using Equation 4.28 to 4.30 it is possible to design the three loops and the index control to the FFT calculation.

To add the cyclic prefix to the symbol, the initial values are stored two times at index n and $n - N_{\text{FFT}}$. The reason for not implementing the cyclic prefix as a cyclic read of the OFDM symbol is that the USRP transmit command does not have that functionality, hence the easiest is to transmit full symbol with cyclic prefix.

4.5.3.3 Hardware Implementation - OFDM

The Cooley Tukey algorithm used for the FFT calculation of the OFDM module offers great possibilities when concerning FPGA implementation regarding e.g. parallel calculations. As seen on the previous Figure 4.7, each radix-2 butterfly calculation can operate as an in place operating, using two memory registers. This means that in each state, no butterfly's input are dependant on another butterfly's output. Inside the butterfly operation, defined in Equation 4.24 to 4.27 some steps in the calculations are dependant on each other, as illustrated in the dataflowgraph in Figure 4.8.

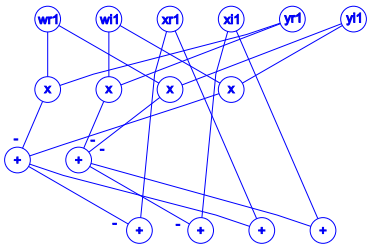


Figure 4.8: Dataflowgraph of the butterfly calculation seen in Equation 4.24 to 4.27.

The dataflowgraph seen in Figure 4.8 can be implemented directly in hardware and it is possible to implement these calculations in parallel for each stage, resulting in the operation usage seen in table 4.6 and Figure 4.9.

Operation	Usage	Calculation time
Addition	$6 \cdot N_{\text{butterflies}}$	$\frac{t_0}{N_{\text{butterflies}}}$
Multiplication	$4 \cdot N_{\text{butterflies}}$	

Table 4.6: Operation usage for the FFT when using the butterfly structure seen in Figure 4.8, with $N_{\text{butterflies}}$ being the number of butterflies and $t_0 = \frac{N_{\text{FFT}}}{2} \log 2 \left(N_{\text{FFT}}\right)$ are the calculation time of a full FFT if only one butterfly was to be implemented.

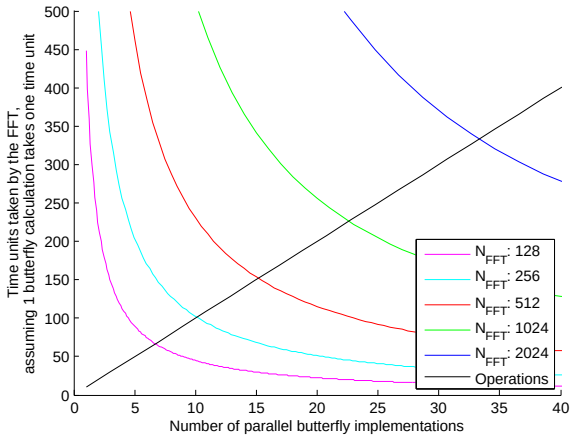


Figure 4.9: The operation usage versus calculation time when implementing parallel butterfly operations in the FFT. Operations are the number of parallel calculations for the given number of parallel butterflies.

As seen on Figure 4.9, the gain from implementing multiple parallel operations fall exponentially, while the resource usage raises linearly. The implementational aspects of the (I)FFT are further analysed in Chapter 5, hence this will not be further examined here.

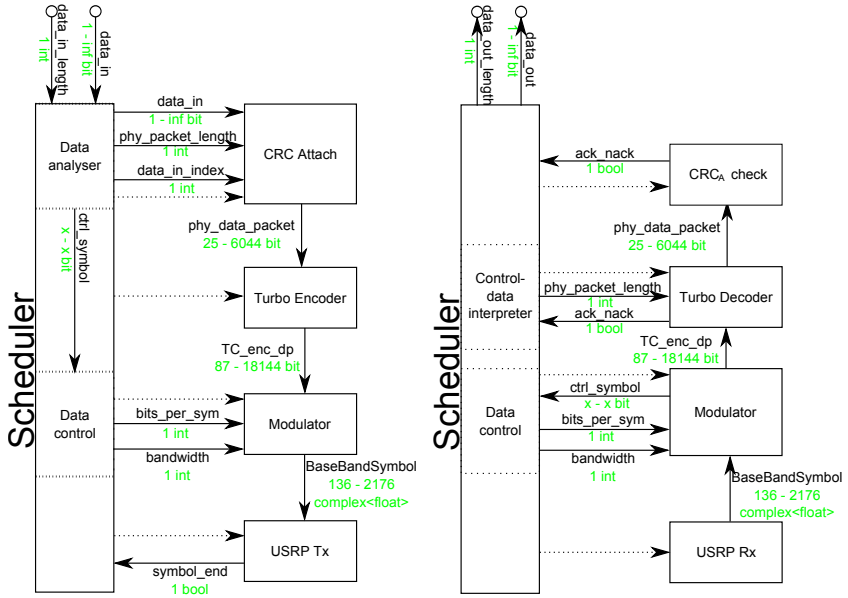
The cyclic prefix is simple to implement, since can be done by a cyclic read of the IFFT.

4.6 C++ program implementation

This section describes the implementation process of all the functions described through this chapter into a complete C++ simulator with purpose of analysing the different functionalities processor use on the Host-PC.

Each functionality described through this chapter has been designed as its own class. A separate transmitter and receiver program has been created such that it is possible to see the resource usage of the different functions independent of each other. The Scheduler in Figure 4.10 is the main program which initializes the different classes and ensures that the data communication between them are as desired. The data connections showed as arrows are different signals which flows in between the classes, whereas a dotted line indicates that the program can signal the function to run.

The *data analyser* is a loop inside the *scheduler* which splits up the generated data into smaller CBs. The *data control* block in the scheduler is created by conditional statements which ensures that the reference symbol and data is written to and read from as desired. Inside the *Scheduler* it is possible to define which and how many resource blocks the system, should use to transmit on.



(a) Transmitter data flow **(b)** Receiver data flow

Figure 4.10: Depicts the flow of the coding and modulation part of the transmitter chain.

As described in Section 3.3.1, the UHD transmit and receive command is blocking calls and it has been chosen to not include them in the analysis program. The UHD transmit and receive commands are not a part of the analysis due to the fact that the Host-PC to/from FPGA communication link always has to be present in this implementation and therefore cannot be altered.

4.7 Algorithm processor analysis

This analysis serves the purpose of analysing the bottlenecks of the C++ program written using the functions described through this chapter. The analysis will be done by the GNU profiler tool *gprof*[41], which is a profiler tool for C++ code which shows the distribution in time used of the function calls in the program.

When analysing the system it is important to have in mind that the actual implementation should not focus on creating a transceiver and not a separate transmitter and receiver. As is stated in Section 4.6 the program which is to be profiled is created as two separate parts, to simplify the design and ease the profiling analysis.

The transmitter and receiver are tested separately by letting the transmitter encode random sequences and storing them in a text file using the C++ library *fstream*. It stores 2 files; *data.dat* which holds the encoded and modulated data

and *original_data.dat* which holds the original data so the received data can be tested with the transmitted.

4.7.1 Analysis results

To gain a complete image of the interrelationship of the different functions the transmitter is tested for different bandwidths (FFT lengths). This helps defining how the load are distributed between, especially, the FFT and turbo decoder, which has been estimated to be the most resource demanding tasks.

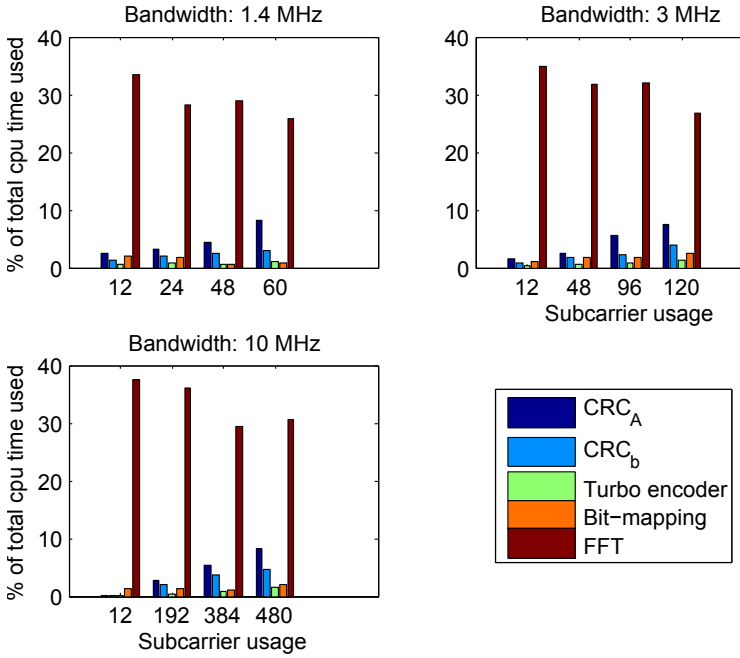


Figure 4.11: Distribution of the total CPU time used over the functionalities implemented in the transmitter.

The first test to be presented is the test of the transmitter. This test is done with different bandwidths, to check if how the FFT complexity has infliction on the usage. Furthermore it is tested with four amounts of subcarrier usage to see how this inflicts the result.

Figure 4.11 shows the trend of the most resource demanding process being the FFT, which was expected since the theory defined the other functions to be of low complexity. It is also seen that the CRC attaching algorithms uses significantly more time than the Turbo encoder, which is due to higher order polynomial. Lastly it is seen that the more subcarriers used, the lower the difference in processor time usage, due to the fact that the fewer symbols has to be transmitted.

The second tests to be presented is of the receiver. These tests operates on the transmitted symbols of the transmitter, and therefore it will test over the same amount of subcarriers and the same bandwidths. Furthermore three

extra tests has been done with different numbers of iterations, to see how this inflicts in the distribution.

The test results of the receiver, shown in Figures 4.12 to 4.14 and clearly shows that the Turbo decoder and FFT are the functions which uses the most of the time in the CPU. While gathering the test results it was noted that after 3 iterations, the CRC_B did use more resources than the CRC_A, unlike in the transmitter. This is due to the fact that it is used as an early stopping algorithm, resulting in it being run one time for each iteration.

The test results of the transmitter states the trend that the more resource block the system are allowed to used for transmission, the less significant the FFT calculation becomes.

It is also noted that the total amount of CPU time spent in the receiver by the functionalities are higher than in the transmitter.

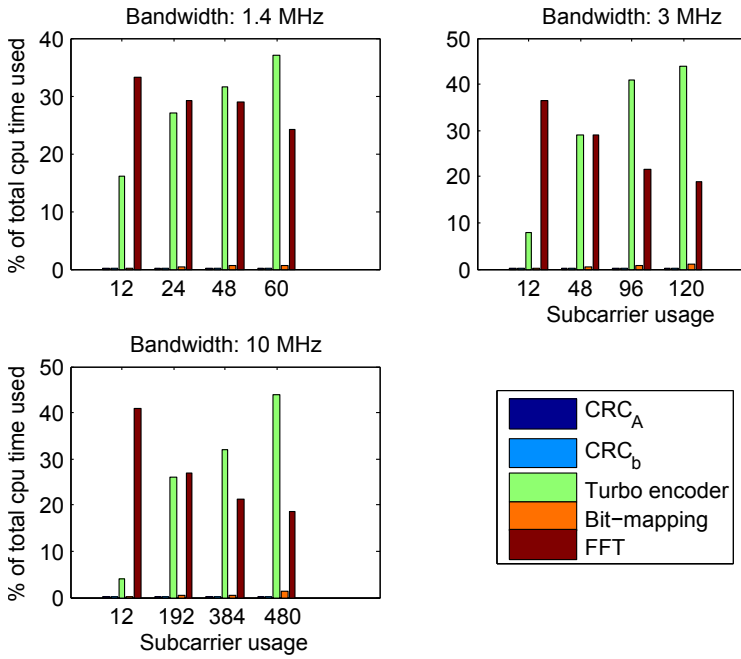


Figure 4.12: Distribution of the total CPU time used over the functionalities implemented in the receiver, using 1 Turbo code iteration.

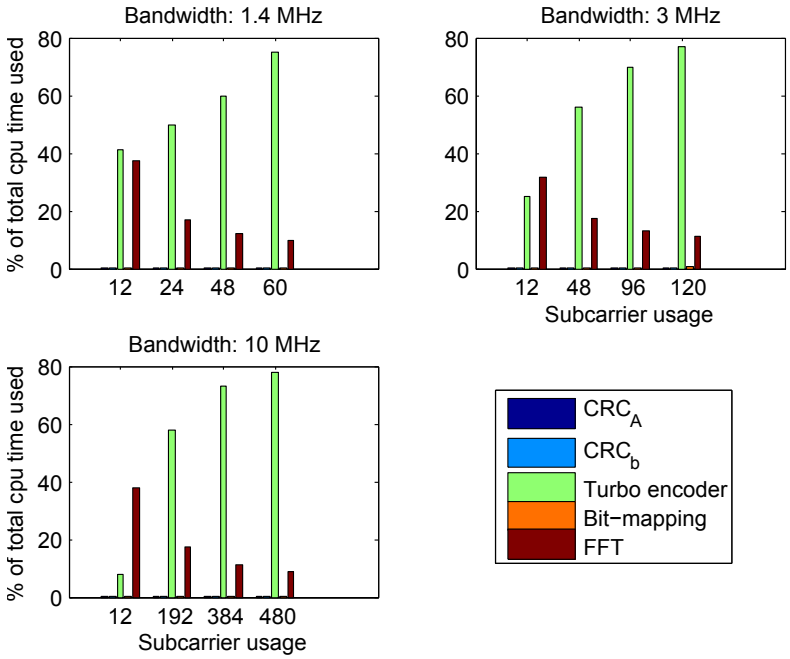


Figure 4.13: Distribution of the total CPU time used over the functionalities implemented in the receiver, using 5 Turbo code iterations.

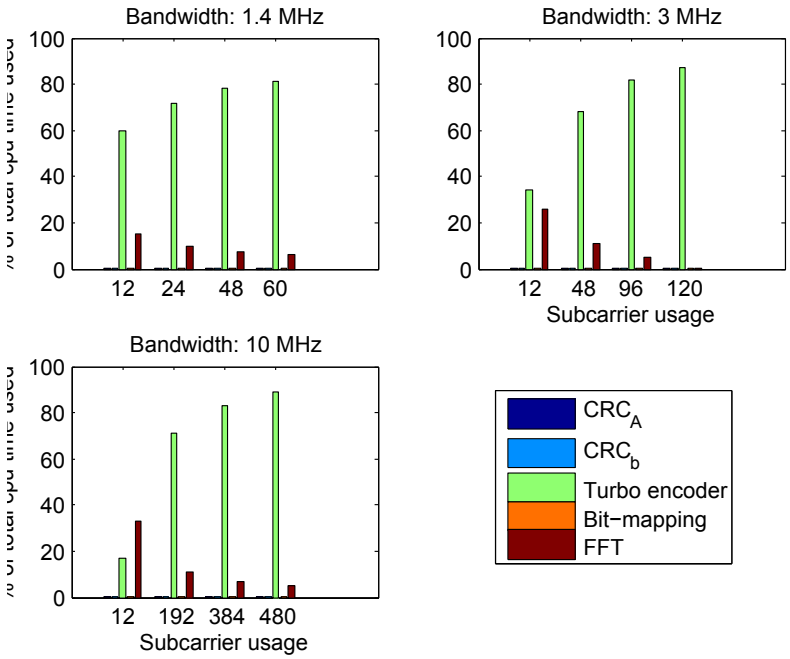


Figure 4.14: Distribution of the total CPU time used over the functionalities implemented in the receiver, using 10 Turbo code iterations.

4.7.2 Implementation Analysis - Summary

As seen from the test results, the most resource demanding functions in the current implementation of a physical layer simulator are the FFT and Turbo decoder, which complies with the analysis of the function complexity done in the current chapter. With an increasing number of iterations, the FFT's resource usage seems to become insignificant. However, the BER performance gain of the Turbo decoder stagnates around 8 iterations [4] and it is very little plausible that the UE is given 50% of the available resources, hence the FFT is still to be a subject of analysis.

The implementation focus of this project was to analyse which functions are tractable to implement on the USRP2, hence the interface between Host-PC and USRP2 must be taken into consideration when determining in which order the implementational focus should be. The current Ethernet connection driver does not naturally support further USRP2 to Host-PC communication other than transmitting and receiving samples, hence these is to be rewritten if any communication should be possible. This is desired to be avoided, since it moves the focus of the project to driver development rather than USRP2 fabric utilization. Due to this, it is decided that The FFT will be the first subject of implementation on the USRP2.

Chapter 5

Hardware implementation

As deduced in the conclusion of Chapter 4, both OFDM and the Turbo decoder can significantly benefit from hardware acceleration, e.g. because they have many calculations which can be done in prallel.

It was evidenced that the FFT which handles the OFDM symbol generation, should be the first subject of implementation because of the following:

- It was not tractable to implement the Turbo decoder on the USRP2 due to its high resource cost and the avaliable time.
- Being the functionality which are the last part of the analysed transmitter chain and first in the analysed receiver chain, implementing the FFT prevents further communication between the USRP and Host-PC. Adding further communication through the Ethernet connection would lead to a lower achieveable datarate and implementation complexity, which is not desired.
- The FFT are the processing demanding functionality of the transmitter and it is one of the high processor functionalities of the receiver.
- The FFT has many possebilities for parallel processing and has been thouroughly researched and optimized.

The following sections of this chapter will consider the design aspects when implementing the FFT on the USRP2, starting with defining the design constraints and then describing the design considerations of the actual modules. A more thourough design description can be found in appendix D.

5.1 Defining the USRP2 constraints

This section covers the preparation made before the desired data processing modules for the FPGA is developed. This includes defining the timing and framework constraints for the design. A description of how the USRP2's dataflow has been analysed are found in Appendix C.

To ensure that the designed data processing modules are compatible with the existing USRP2 FPGA image's functionality, it has been chosen that the module designed module should be transparent; e.g. it should adapt the signal flow from where it is placed, without the need to alter in any other parts of the FPGA code. Section 3.4 deduces that the module should be implemented in between the *dsp_core* and the *vita_control* module due to the fact that it is possible to intercept and further data process the samples here.

Based on the signal analysis in Appendix C, a structural template for the transparent data processing module is created as seen in Listing 5.1 with the inputs and outputs described in Table C.1.

```
1  module data_process
2    #(parameter MEM_WIDTH = 1280,
3      parameter setting_reg_BASE = 160,
4      parameter L2_PACKET_WIDTH = 7,
5      parameter SAMPLE_RES = 32)
6    (input clk,
7     input reset,
8     input clear,
9     input run,
10    input [31:0] sample,
11    input strobe_i,
12    input [L2_PACKET_WIDTH-1:0] samples_pr_packet,
13    input [L2_PACKET_WIDTH:0] desired_fft_length,
14    output wire [31:0] sample_o,
15    output wire strobe_o);
```

Listing 5.1: The template of the module which was created in Appendix C based on the signal analysis that was made.

This template was designed to hold all signals required to create a module which could work coherently with the existing FPGA image. The structure of the data processing module, and its sub modules which are utilizing this template are thouroughly described in Appendix D.

With the constraints of the system defined, it is possible to design the required data processing module.

5.2 Data processing modules

This section serves the purpose of documenting the modules designed through this project. A more thourough description are avaliable in Appendix D.

As in Appendix D, the submodules of the *data_processing* module will be described in the order they are implemented, illustrated by Figure 5.1.

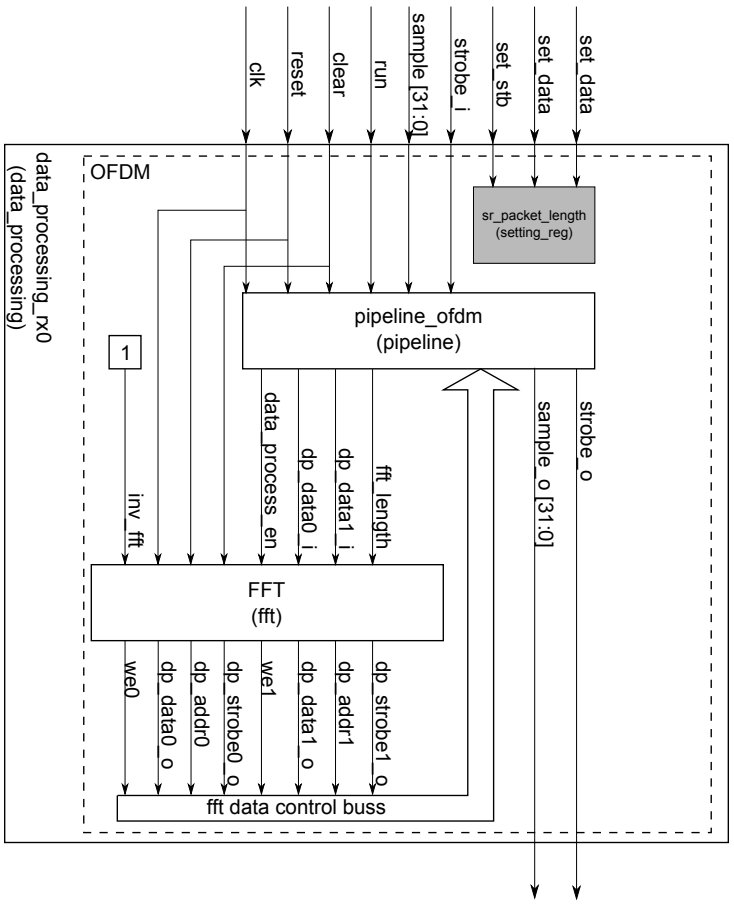


Figure 5.1: Illustration of the submodules and their function calls. As can be seen, the main module *data_processing*, which uses the template defined in Appendix C, is the top module. The grey boxes are reused from ETTUS research and will not be described in this chapter.

5.2.1 Pipeline module

Due to the incompatability between the serial dataflow of the USRP2 FPGA image and the packet based data processing of the LTE technologies, a pipeline module is created. This module serves the purpose of collecting incoming data samples in packages which can then be data processed by the *fft* module.

This section will describe the *pipeline_2N* module with $N_{\text{packets}} = 2$, which yeilds a minimal delay for the data collecting to the data output. It is possible to reuse the base of this module code and adding a greater delay to the pipeline, hence the pipeline modules are named accordingly to their pipeline depth, eg. *pipeline_2N* module does contain two packets. A length of two will results a time avaliable for data processing $t_{\text{data processing}}$ to be equal to how long it takes to fill one memory registers.

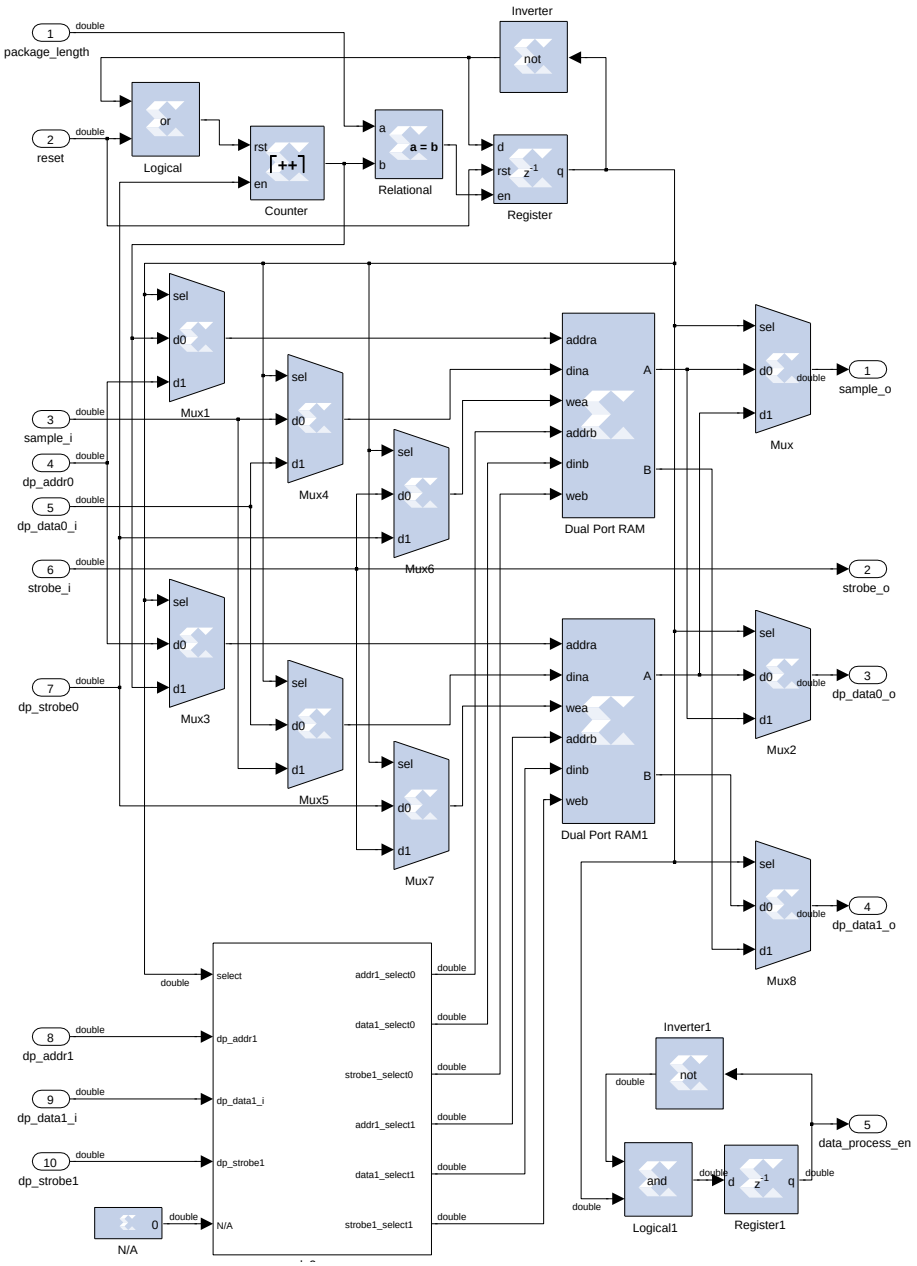


Figure 5.2: Illustration of the *pipeline_rx* modules structure, using $N_{\text{packets}} = 2$.

Figure 5.2 are the block diagram describing the pipeline module with $N_{\text{packets}} = 2$. As seen, the pipeline module mostly consists of ram blocks and different multiplexers, where the different signals are more thouroughly described in Appendix D. As described in Appendix D, the memory block used in the *pipeline_2N* module are a reused module from ETTUS research called *ram_2port*.

The reuse was done to ensure that the ram blocks were compatible with the compiler options of the USRP FPGA image.

Through the design process two different approaches of designing the signal flow was tested:

1. The first design delayed the data sample output of the actual USRP2 dataflow by utilizing some of the control signals used in the existing FPGA image. As described in section D this was not possible without alternating some of already existing modules and possible the UHD on the host-pc, hence this was discarded.

2. The design of the *pipeline_2N*'s dataflow was instead created such that it does output a sample each time the *dsp_core* module does, however the first $2N$packets samples are NULL. This will make the use of the *data_process* module less transparent in a C++ implementation, however it will increase the transparency in respect to implementation and usage with the USRP.

5.2.1.1 Test

Different tests of the codes functionality was done using a testbench written in verilog and debugged using the Xilinx ISim GUI[1]. Appendix D shows a more thourough description of the test.

The USRP test was done by implementing the pipeline module directly in the USRP2's FPGA image, as seen in Listing D.3, Appendix D. When the FPGA image was compiled it was tested if any throughtput was possible, using the test setup seen in Figure 5.3.

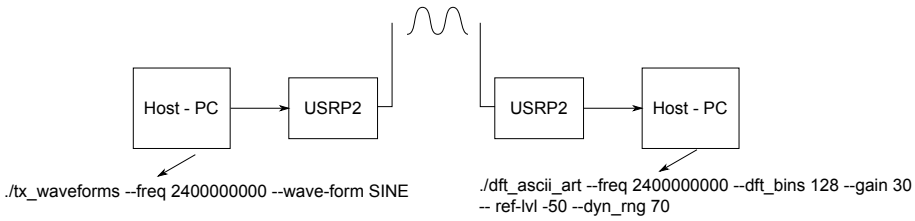


Figure 5.3: Test setup to see if the implementation of the *pipeline_2N* module still supports the original UHD examples. The commands under the host pc's are the commands used in the Linux terminal on the given PC.

The results of the test was that the pipeline module did insert a small delay, visible since the output of the DFT on the host PC was 0 for a short period of time. However, after the initialization of the memory registers the *pipeline_2N* module was transparent as desired, since it did not interfere with the output.

[1]Graphical User Interface

The resource usage of the *pipeline_2N* module was estimated by the Xilinx ISE Design Suite synthesizer. A table of the results of this can be seen in Table 5.1.

Logic Utilization	Used	Available	Utilization
Number of Slice Flip Flops	48	40,960	1%
Number of 4 input LUTs	285	40,960	1%
Number of occupied Slices	182	20,480	1%
Total Number of 4 input LUTs	295	40,960	1%
Memory usage	81920	720000	11.4%

Table 5.1: The table shows the *pipeline_2N* modules utilization of the Spartan 3 FPGA on the USRP2.

$$N_{\text{Slices}} = \frac{81920 - (0.03 \cdot 720000)}{2} = 30160$$
$$\text{Fabric} = \frac{30160 + 182}{20480} = 148\% \tag{5.1}$$

The resource usage of the *pipeline_2N* module is quite low. It only uses 182 slices, which is only 1% of the fabric and since 58% is free this is more than sufficient. However, due to the low amount of available memory, the slice resource usage increases to 148 % of the available, since the memory has to be moved to the fabric. Through the deduction of the code, it was found that it is possible to free some RAM by optimizing the functions made by Ettus research, however the result implies that it would be practical to use the USRPN200 for implementation.

5.2.2 *FFT* module

The *fft* module is implemented by using the Cooley Tukey algorithm, which is briefly summarised in Section 4.5.3.1. This algorithm is based on a certain amount of in-place calculations called "butterflies" which can have different number of inputs, described by the "radix". Using a high radix will result in the possibility of simplifying the calculations. However, a high radix will mean that the FFT will not support all sizes in the range 2^n for $n = [1, 2, ...\infty]$, as is the case with a radix-2 implementation, as seen in Table 5.2.

Radix-2	Radix-4	Radix-8	Radix-16
16	16		16
32			
64	64	64	
128			
256	256		
512			
1024	1024	1024	1024
2048			
4096	4096		

Table 5.2: Which FFT lengths are supported by different radix's of the FFT, within the area of the LTE bandwidth (128 - 2048).

To ensure that the *fft* module can support any length required, it has been chosen to use the radix-2 butterfly algorithm. In Appendix D it has been deducted that the desired calculation should proceed in two sequential calculations.

$$y_{temp0} = -w_r \cdot y_r + w_i \cdot y_i \tag{5.2}$$
$$y_{temp1} = -w_r \cdot y_i - w_i \cdot y_r \tag{5.3}$$
$$X_r = x_r - y_{temp0} \tag{5.4}$$
$$X_i = x_i - y_{temp1} \tag{5.5}$$
$$Y_r = x_r + y_{temp0} \tag{5.6}$$
$$Y_i = x_i + y_{temp1} \tag{5.7}$$

Equation 5.2 to 5.3 shows the first step of the sequential calculation, which calculates two temporary values to be used in the second step. According to Appendix D this intermediate result is to be calculated to reduce the resource usage by two additions. Equation 5.4 to 5.7 show the second step of the butterfly calculation, which uses the intermediate values.

Through the timing analysis in section D it was decided that the initial design should focus on implementing a FFT of length 1024, utilizing the dual port ram of the *pipeline_2N* module to create a full radix-2 butterfly calculation each clock cycle. Even though this focus of implementation results in the initial design to only support a bandwidth of 10 MHz, it will lower the required control logic significantly and the result will show if it is a tractable solution to implement a FFT supporting the 20 MHz bandwidth.

Since the *ram_2port* module used in the *pipeline_2n* uses one clock cycle to load and one to store the data, the design of the *fft* module were optimized such that it uses two parallel radix-2 algorithms to calculate a 4 point FFT, using the flow seen on Figure 5.4.

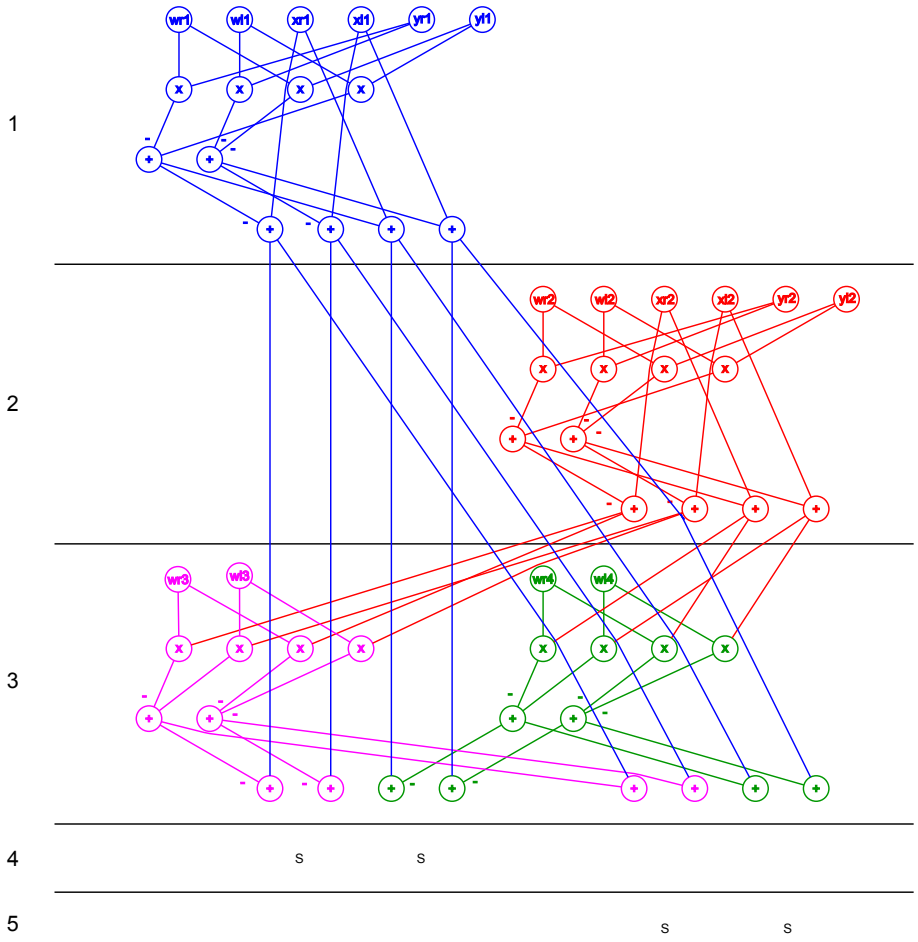


Figure 5.4: The final dataflow of the 4-point (I)FFT used in this project. This graph has been created based on an analysis of Equation 4.24 to 4.27.

The basic design strategy of Figure 5.4 is to reduce the amount of memory load/write operations, compared to when using a single radix-2 operation.Even though this design results in a structure which seems as a radix-4, it is possible to cover all bandwidths by using an extra loop using only one stage of radix-2 operations.

The final steps taken before the verilog implementation of the *fft* module was to define the in- and outputs of the module, as illustrated in Figure 5.5, and defining which submodules that were needed and how the interconnection of these should be.

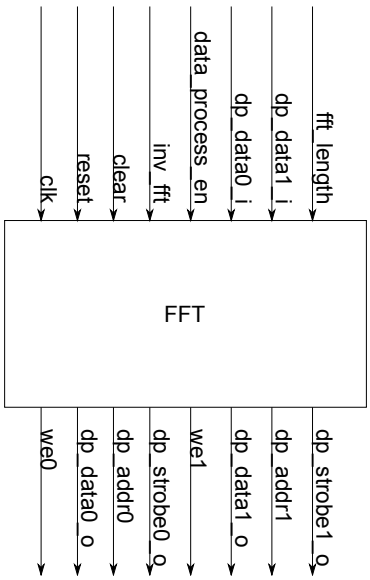


Figure 5.5: In- and outputs of the FFT module. As seen the inputs consists of some control signals together with two data in ports which are used to read from the dual access ram in the pipeline. The outputs are pure ram access signals for reading and storing in the pipeline ram.

5.2.3 Sub Module Design and Implementation

This section briefly covers the submodules used in the *fft* module and how these are connected. The internal design of the *fft* module is seen in Figure 5.6. If a more thorough description of each submodule is desired; refer to Appendix D.

- *Ext. RAM* are the external RAM block in the *pipeline_2N* module which holds the data to be processed.
- The *control_logic* is the control signal dataflow for each 4-point FFT calculation, written as code in the main *fft* module. The *control_logic* flow controls, among other things, the assertion of *do_stb*, *input_strobe* and *stb_strobe*. It also controls the output of the muxes which decides wether to use internal or external memory in the calculation.
- The *data_addr_calc* module controls the address output which are used to access the external RAM block. The module takes input which describes how far in the FFT the system is and outputs addresses which corresponds to the bit reversed input of the desired data index.
- The function of *input_pair_ctrl* module can be conceived as a FSM which increases the value of the outputs in a speficic order. A new output is requested every time a radix-2 butterfly calculation is begun, thus updating the control signals for the next radix-2 butterfly calculation while calculation the current.
- The *twiddle_addr_count* and *twiddle_rom* modules calculate the twiddle factors for the given radix-2 butterfly calculation. This is done by *twid-*

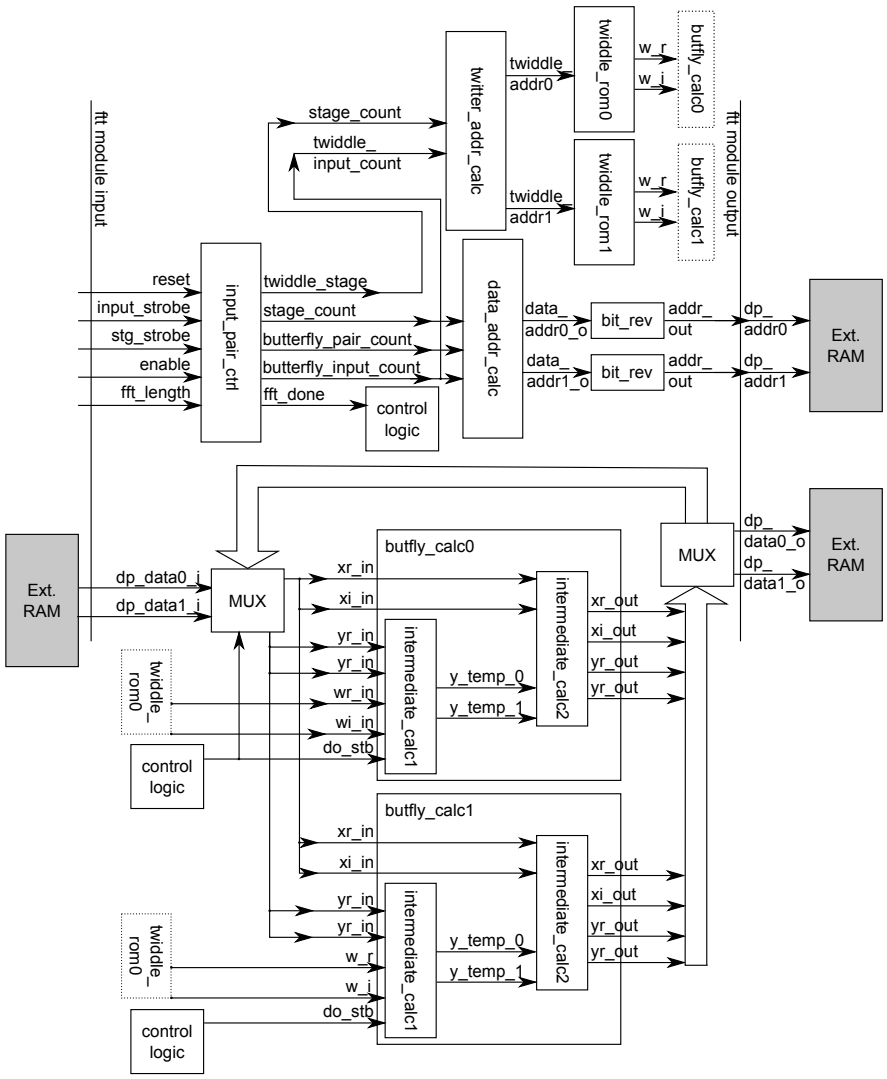


Figure 5.6: The interconnections of the different submodules in the *fft* module.

dle_addr_count first calculation the correct twiddle addresses from the current *stage_count* and *butterfly_input_count*. Then *twiddle_rom* takes these addresses and finds the corresponding twiddle factors in the lookup table.

- The *butfly_calc* module handles the radix-2 calculation. The calculation are done in half a clock cycle, and initialized on the *do_stb* using four multipliers and four additions.
- The *bit_rev* uses routing to make the output address bit reversed as desired.

The program flow is designed as seen in Listing D.8.

5.2.3.1 Test of the *FFT* module

Before the *fft* module is to be implemented in the *data_process* module, it is tested with two purposes in mind, one being to check if it can actually calculate an FFT and the second is to see the resulting resource usage, which can be used to see if it is plausible to implement it on the USRP.

To test if the *fft* module works as desired a testbed has been made. This testbed is seen in the *fft_tb* module and it emulates the input signals to the FFT, which come from the *pipeline_n2* module. The input data is loaded from a txt file and the output is stored in the RAM. The outputs in the RAM are compared to the output of a Matlab calculated fft with the same inputs as in the txt file. The comparison revealed that the FFT was calculated correctly.

To find the resource usage, the synthesizer in the Xilinx ISE Design Suite has been run. The results of this can be seen in Table 5.3.

# ROMs[2]	: 4
512x15-bit ROM	: 4
# Multipliers	: 9
10x10-bit multiplier	: 1
16x16-bit multiplier	: 4
31x16-bit multiplier	: 4
# Adders/Subtractors	: 22
10-bit adder	: 1
11-bit adder	: 2
16-bit adder	: 4
17-bit adder	: 4
17-bit subtractor	: 4
31-bit adder	: 4
31-bit subtractor	: 2
4-bit adder	: 1
# Registers	: 65
Flip-Flops	: 65
# Latches	: 2
10-bit latch	: 2
# Logic shifters	: 3
11-bit shifter logical left	: 2
11-bit shifter logical right	: 1

Table 5.3: The table shows the resource usage of the *fft* module.

Logic Utilization	Used	Available	Utilization
Number of 4 input LUTs	1053	40,960	2.6%
Number of occupied Slices	535	20,480	2.6%
Number of Slices containing only related logic	535	535	100%
Number of Slices containing unrelated logic	0	535	0%
Total Number of 4 input LUTs	1053	40,960	2.6%
Memory usage	30720	720000	4.2%

Table 5.4: The table shows the *fft* modules utilization of the Spartan 3 FPGA on the USRP2.

$$N_{\text{Slices}} = \frac{30720 - (0.03 \cdot 720000)}{2} = 4560$$
$$\text{Fabric} = \frac{4560 + 535}{20480} = 24.8\% \tag{5.8}$$

It can be concluded, that only a small portion of the FPGA fabric is used, as seen in Table 5.4 only about 2.6 % are used and only 9 multipliers are used. Since there were 58% of the fabric available and only 2.6% is needed. Combined with the fact that only 9 multipliers are used, which amounts to 22% of the multipliers were 57% are available, it seems plausible to implement the FFT on the USRP2. However, it should be noted that the amount of memory used in the system exceeds the available amount by a little over 1%, so about 1140 CLBs of the fabric has to be used for memory storage as seen in Equation 5.8. Although this is means the 24.8% of the fabric to use on memory, the FFT itself is still considered implementable.

5.3 Module Implementation

This section describes the steps taken when implementing the modules described though the chapter into the *data_process* module.

From Section 5.2.1 it is seen that the pipeline resource usage implies that the avaliable fabric and memory on the USRP2 is not sufficient.

This means that if the OFDM calculation should be implemented sucessfully, it has to be on the USRPN200. However, when transferring the code to the other platform, it was seen that some of the modules did not comply with the compiler. Due to lack of time, this resulted in the fact that no tests of the *data_process* module was made when it was implemented on the USRPN200 or USRP2.

Chapter 6

Conclusion

Through the report, the functionalities of LTE's physical layer has been addressed concerning the motivation and implementational issues. It was found that an USRP2 would be an excellent platform to develop hardware acceleration algorithms on, due to the fact that it already is a working SDR platform which is utilized by many. The USRP2 also has available resources for further hardware implementations.

The analysis of the implementational issues were conducted by addressing possible implementation methods on a CPU and FPGA firmware. By first implementing the functions on a CPU, it was possible to see the distribution of resource usage by the different functionalities, which, when combined with the analysis of possible HW implementation gain such as pipelining, gave an insight in which functionalities that were tractable to implement on the USRP2 platform. The implementation on the CPU was done in C++ using single core program options.

6.1 Findings

The primary focus of the implementational aspect was found to be the OFDM modulation using a Cooley Tukey (I)FFT. This was concluded since it was high on the resource usage table for the initial C++ algorithm and it offered a high possibility of gaining from hardware acceleration due to possible pipelining steps. Furthermore, it was also deduced that implementation of modules elsewhere in the chain would result in an increased Host-PC to USRP2 communication load, which was not possible. An increase in Host-PC to FPGA communication would also result in the fact that some of the UHD firmware would need to be rewritten, which would make the implementation of a new module more exhaustive.

The Turbo decoder has the possibility of gaining a lot from hardware acceleration but it was deduced that it would not be a tractable solution due to the available USRP2's resources and speed. However, it would be possible if the expansion port on the USRP2 was used, since a faster and more resourceful FPGA could be connected.

The first thing that was designed and implemented successfully was a module which collected a predefined number of data samples into packets, which could be data processed. The module was called *pipeline_2N* because it was a pipeline consisting of two memory blocks. These two memory blocks were inserted such that one datablock could be data processed while the other one was receiving data.

The *pipeline_2N* module was tested and found to be working in correspondence with the rest of the FPGA image. However, the two first packets of the new data processing module are to be discarded from the receiver due to the nature of the rest of the FPGA image.

Initial analysis of the FFT implementation showed that it would be a challenge to implement it, due to the fact that the relatively slow 100 Mhz clock on the USRP2 together with the maximum FFT length in LTE being 2048, would result in only 0.63 available clock cycles for each butterfly (Appendix D) in the FFT. The available number of clock cycles would mean that there would have to be calculated up to 32 parallel butterflies, which would require up to 128 multipliers, without optimization. This was not tractable since only 23 dedicated multipliers are available on the USRP2.

It was decided that the focus would be on actually implementing a working FFT on the USRP2 within the timeframe. To obtain this, the FFT length was reduced to 1024 which resulted in 1.34 available clock cycles for each butterfly operation. With 1.34 available clock cycles per butterfly, only 4 parallel butterflies would be needed, resulting in a maximum of 16 multiplications used.

Further analysis, using a dataflowgraph, concluded that a radix-4 algorithm could be created with the desired speed and use only two butterfly operations, resulting in a total use of only 9 multiplications.

It was shown that the FFT would use up to 2.6 % of the Spartan 3 fabric, but when implementing it in correspondence with the existing USRP2 FPGA image this increased to 24.8 % due to the small remaining amount of avaliable memory on the USRP2 (3 %). However, it would still be possible to implement the *fft* module.

Through the implementation process, it was concluded that the low amount of remainding memory on the USRP2 was not sufficient to implementation of the *data_process* module, due to the combined usage of the *pipeline_n2* and *fft* module.

6.2 Future Work

Due to the fact that the amount of memory required for the OFDM system exceeded the remainding on the USRP2, the implementation was not completed. This should however be easily done by changing the platform to the USRPN200, since this has the double amount of memory avaliable. It was however noted that some of the modules written did not fully comply with the USRPN200 FPGA image, hence some revising must be done.

Another implementational issue which should be addressed is that the current design does have a hard coded bandwidth and hence only supporting

one bandwidth for each compilation. To circumvent this, the created module should calculate the desired bandwidth based on the Ethernet packet length set by the user on the Host-PC. This can easily be done by using the setting register which holds the Ethernet packet length information and data process this.

The next step for implementation should be the bit (de)mapping implementaion, which can be done very cheap by the proposed structure.

Bibliography

[1] E. Network and I. S. Agency. [Online]. Available: http://www.enisa.europa.eu/activities/application-security/smartphone-security-1/top-ten-risks/network-congestion

[2] [Online]. Available: http:\www.3gpp.org

[3] M. Q. Manuel Uhm. (2012) Xilinx. [Online]. Available: http://www.wirelessdesignmag.com/ShowPR.aspx?PUBCODE=055&ACCT=0029861&ISSUE=1002&RELTYPE=ic&PRODCODE=R0190&PRODLETT=A&CommonCount=0

[4] B. Vucetic and J. Yuan, *Turbo Codes: Principles and applications*. Kluwer Academic Publishers, 2000.

[5] S. S. G. Tenkasi V Ramabadran, *A Tutorial on CRC Computations*. IEEE MICRO, 1988.

[6] *Physical Channels and Modulation (Release 8) TS 36.211*, 3GPP Std.

[7] C. Berrou, A. Glavieux, and P. Thitimajshima, "Near shannon limit error - correcting coding and decoding : Turbo-codes (1)," p. 7, 1993.

[8] B. Sklar, "A primer on turbo codes concepts," 1997.

[9] J.-F. . T. Cheng and H. Koorapaty, "Error detection reliability of lte crc coding," p. 5, 2009.

[10] J.-F. T. Cheng, "Two-level early stopping algorithm for lte turbo decoding," p. 5, 2008.

[11] T. Jessen, R. Simonsen, M. Buhl, and J. L. Buthler, "Turbo codes and ofdm implementation for lte mobile systems," 8th semester report, AAU 2011.

[12] *Multiplexing and channel coding (Release 9) TS 36.212*, 3GPP Std.

[13] *Overview of 3GPP Release 8 V0.2.3*, 3GGP Std.

[14] A. Nimbalker, T. K. Blankenship, B. Classon, T. E. Fuja, and D. J. Costello, "Contention-free interleavers for high-throughput turbo decoding," 2008.

[15] A. Nimbalker, T. E. Fuja, D. J. C. Jr, T. K. Blankenship, and B. Classon, "Contention-free interleavers."

[16] F. Tosato and P. Bisaglia, "Simplified soft-output demapper for binary interleaved cofdm with application to hiperlan/2," p. 6, 2001.

[17] J. Lee, H.-L. Lou, D. Toumpakaris, and J. M. Cioffi, "Snr analysis of ofdm systems in thepresence of carrier frequency offset for fading channels," 2006.

[18] S. Signell, "Ofdm systems - why cyclic prefix?" 2008.

[19] B. E. Priyanto, "Air interfaces of beyond 3g systems with user equipment hardware imperfections: Performance & requirements aspects a case study on utra long term evolution uplink," Ph.D. dissertation, Aalborg University, 2008.

[20] H. G. Myung, J. Lim, and D. J. Goodman, "Single carrier fdma for uplink wireless transmission," *IEEE Vehicular Technology*, sept 2006.

[21] M. Inc., "Technical white paper : Long term evolution (lte): Overview of lte air-interface," Motorola, Tech. Rep., 2007.

[22] S. Haykin, *Adaptive Filter Theory*, T. Kailath, Ed. Patience Hall inc., 1995.

[23] S. A. Ltd., "Improving throughput performance in lte by channel estimation noise averaging," p. 3, 2010.

[24] L. Somasegaran, "Channel estimation and prediction in umts lte," 2007.

[25] R. Koetter, A. C. Singer, and M. Tchler, "Turbo equalization: An iterative equalization and decoding technique for coded data transmission," p. 14, 2004.

[26] M. Tchler, R. Koetter, and A. C. Singer, "Turbo equalization: Principles and new results," 2002.

[27] M. Tchler and A. C. Singer, "Turbo equalization: An overview," 2011.

[28] Q. Gu, *RF System design of transceivers for wireless communication.* Springer, 2006.

[29] [Online]. Available: https://www.ettus.com/product/category/Daughterboards

[30] Xilinx, *Spartan-3 Generation FPGA User Guide*, 1st ed., 6 2011.

[31] [Online]. Available: http://www.ettus.com/faq

[32] [Online]. Available: http://www.xilinx.com/technology/dsp/xtremedsp.htm

[33] S. V. Enevoldsen, J. G. Johansen, and V. Pucci, "Analysis and architectural mapping of an fft algorithm into an already existing fpga firmware of a low-cost cots sdr peripheral," Aalborg University, Tech. Rep., 2011.

[34] M. May, T. Ilnseher, and N. Wehn, "A 150mbit/s 3gpp lte turbo code decoder," p. 9, 2010.

[35] C. Studer, C. Benkeser, S. Belfanti, and Q. Huang, "Design and implementation of a parallel turbo-decoder asic for 3gpp-lte," vol. 46, p. 10, 2011.

[36] W.-P. Ang and H. K. Garg, "A new iterative channel estimator for the log-map & max-log-map turbo decoder in rayleigh fading channel," p. 5, 2001.

[37] L. Li, R. G. Maunder, B. M. Al-Hashimi, and L. Hanzo, "A low-complexity turbo decoder architecture for energy-efficient wireless sensor networks," p. 9, Na.

[38] M. A. Bickerstaff, D. Garrett, T. Prokop, C. Thomas, B. Widdup, G. Zhou, L. M. Davis, G. Woodward, C. Nicol, and R.-H. Yan, "A unified turbo/viterbi channel decoder for 3gpp mobile wireless in 0.18-μm cmos," p. 10, 2002.

[39] M. Bickerstaff, L. Davis, C. Thomas, D. Garrett, and C. Nicol, "A 24mb/s radix-4 logmap turbo decoder for 3gpp-hsdpa mobile wireless," p. 10, 2003.

[40] F.-M. Li, C.-H. Lin, and A.-Y. Wu, "Unified convolutional/turbo decoder design using tile-based timing analysis of va/map kernel," vol. 16 Issue 10, p. 5, 2008.

[41] [Online]. Available: http://www.cs.utah.edu/dept/old/texinfo/as/gprof_toc.html

List of Figures

List of Tables

Appendix A

FPGA resource tables

Directory	Content
boot_cpld	CPLD bootloader firmware
control_lib	Various control structures, RAM primitives and FIFO[1] buffers
coregen	FIFO buffers based on Xilinx generators
extram	External memory controller
extramfifo	External memory FIFO
fifo	Various FIFO buffers
gpmc	General Purpose Memory Controller utilities (not used in USRP2_Rev2)
models	Various logic primitives
opencores	ZPU and various peripheral controllers from the opencores.org project
sdr_lib	Signal processing modules. Contains dsp_core modules
serdes	Contains flow-control logic for the SERDES interface
simple_gemac	Gigabit Ethernet MAC modules
testbench	Testbench support code
timing	Timer/counter modules
top	Top-Level modules
udp	UDP transmit and receive engine
vrt	VITA Radio Transport protocol code

Table A.1: The different directories of the FPGA firmware hierachy and which modules they contain[33, 8].

Appendix B

Soft-output demodulation

This appendix clarifies how the different modulation schemes are defined in LTE and a soft-output demodulator are designed. In LTE a Gray encoded[1], signal constellation of BPSK, QAM, 16-QAM and 64-QAM are defined[6].

The different constellations are illustrated in the corresponding section by a MATLAB plot. For the exact values of the symbols refer to Equations 4.17 to 4.19 or the standard [6].

B.1 BPSK

The BPSK constellation described in LTE can be seen on figure B.1.

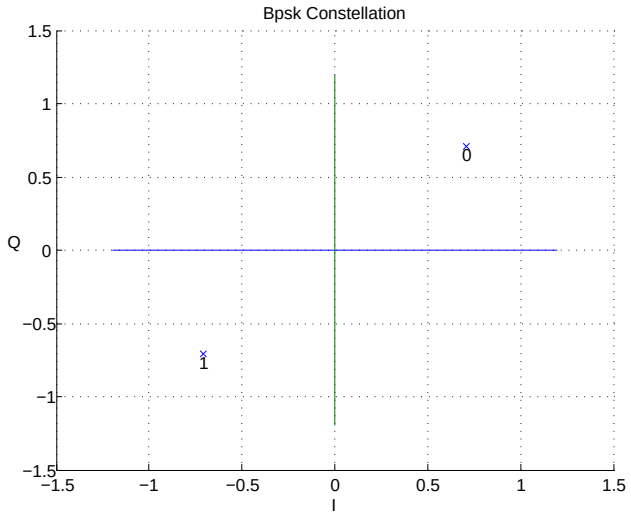


Figure B.1: BPSK constellation as defined in the LTE.

[1] In Grey encoding the hamming distance between a symbol and it's neighbouring symbols is 1

As seen on Figure B.1 the two bits are described by a skewed placement of the bits. However, to illustrate the decoding principle a non skewed constellation will be used as the one described in Equation B.1.

Given a binary message signal $m(k) \in \{0, 1\}$, the BPSK modulated signal to be transmitted $s(k)$ can be described by equation B.1.

$$s(k) = (-1)^{|m(k)-1|} \tag{B.1}$$

The baseband signal $s(k)$ are in this case analysed to be transmittet through an AWGN channel. Since the modulated signal are real only the noise $z_{\text{AWGN}}(k)$ with mean μ_z and noise power σ_z^2 are defined to be aswell. The received signal $\tilde{s}(k)$ is defined as seen in Equation B.2.

$$\tilde{s}(k) = s(k) + z(k) \tag{B.2}$$

Due to the additive noise, the received signal will be randomly distributed as illustrated in figure B.2. This normal distribution is specified with a mean and a variance. The mean depends on which modulation scheme is used and the variance depends on the noise in the channel and the signal strength.

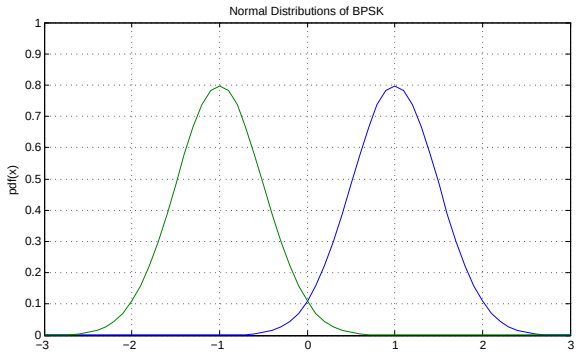


Figure B.2: Illustration of the probability distribution of the received symbol $\tilde{s}(k)$, assuming an AWGN channel with $\mu_n = 0$ and $\sigma_n^2 = 1$.

As seen on Figure B.2, the two distributions overlap each other, resulting in that the receiver has to analyse the received signal before deciding. In order to evaluate what the received binary message signal should be deciphered as, the ML decision rule is used, illustrated in equation B.3:

$$\tilde{m}(k) \begin{cases} 1 & \text{for } \frac{\mathrm{P}(m=+1 \mid \tilde{s}(k))}{\mathrm{P}(m=-1 \mid \tilde{s}(k))} > 1 \\ 0 & \text{for } \frac{\mathrm{P}(m=+1 \mid \tilde{s}(k))}{\mathrm{P}(m=-1 \mid \tilde{s}(k))} < 1 \end{cases} \tag{B.3}$$

Where $\mathrm{P}\left(m = -1 \mid \tilde{s}(k)\right)$ is the probability of the received signal $\tilde{m}(k)$ being -1, given the received symbol $\tilde{s}(k)$. With $s(r)$ being the possible values of the transmittet signal $s(k)$, the probability is defined as equation B.4.

$$\mathrm{P}\left(m(k) = s(r) \mid \tilde{s}(k)\right) = \frac{1}{\sqrt{2 \cdot \pi \cdot \sigma_n^2}} \cdot \exp\left(-\frac{(\tilde{s}(k) - s(r))^2}{2 \cdot \sigma_n^2}\right) \tag{B.4}$$

The bit decision in equation B.3 are called "hard decision", since the output will always be consistent with the alphabet of s.

In a communication system there are typically incorporated some kind of ECC[2] to increase the BER of the received stream. Some of these codes, as Turbo Codes described in Section 2.2, can increase the BER by processing the bit probability instead of the hard decision. Given that the probabilities have exponential characteristics, it can be beneficial to use the LLR[3] of the bit instead, as defined in Equation B.7 for BPSK with $s(r) \in (-1, 1)$.

$$\ln\left(\frac{P(\tilde{m}(k)=1 \mid \tilde{s}(k))}{P(\tilde{m}(k)=-1 \mid \tilde{s}(k))}\right) = \ln\left(\frac{\frac{1}{\sqrt{2\cdot\pi\cdot\sigma_n^2}}\cdot\exp\left(-\frac{(\tilde{s}(k)+1)^2}{2\cdot\sigma_n^2}\right)}{\frac{1}{\sqrt{2\cdot\pi\cdot\sigma_n^2}}\cdot\exp\left(-\frac{(\tilde{s}(k)-1)^2}{2\cdot\sigma_n^2}\right)}\right) \tag{B.5}$$

$$= \ln\left(\exp\left(-\frac{(\tilde{s}(k)+1)^2}{2\cdot\sigma_n^2}\right)\right) - \ln\left(\exp\left(-\frac{(\tilde{s}(k)-1)^2}{2\cdot\sigma_n^2}\right)\right) \tag{B.6}$$

$$= -\frac{(\tilde{s}(k)+1)^2}{2\cdot\sigma_n^2} + \frac{(\tilde{s}(k)-1)^2}{2\cdot\sigma_n^2} \tag{B.7}$$

When the modulation order (the alphabet of s) increases, the complexity of the decoding process increases aswell. The next section will describe how it is possible to decode a higher modulation order system.

B.2 QAM

The QAM modulation order used in this report are Grey encoded as seen on figure B.3 and can be described by equation B.8.

Since there are 2 bits per symbol, it is assumed that m are double the size of s.

$$s(k) = \left((-1)^{|m(2k)-1|} + \mathrm{j}(-1)^{|m(2k+1)-1|}\right) \cdot 1/\sqrt{2} \tag{B.8}$$

Due to the Grey encoding it is possible to analyse each of the two bits seperatly, using the real and imaginary value of the symbol, treating it like two seperate BPSK signals.

$$\tilde{m}(2k) = -\frac{(\tilde{s}_{\mathrm{real}}(k)+1)^2}{2\cdot\sigma_n^2} + \frac{(\tilde{s}_{\mathrm{real}}(k)-1)^2}{2\cdot\sigma_n^2} \tag{B.9}$$

$$\tilde{m}(2k+1) = -\frac{(\tilde{s}_{\mathrm{imag}}(k)+1)^2}{2\cdot\sigma_n^2} + \frac{(\tilde{s}_{\mathrm{imag}}(k)-1)^2}{2\cdot\sigma_n^2} \tag{B.10}$$

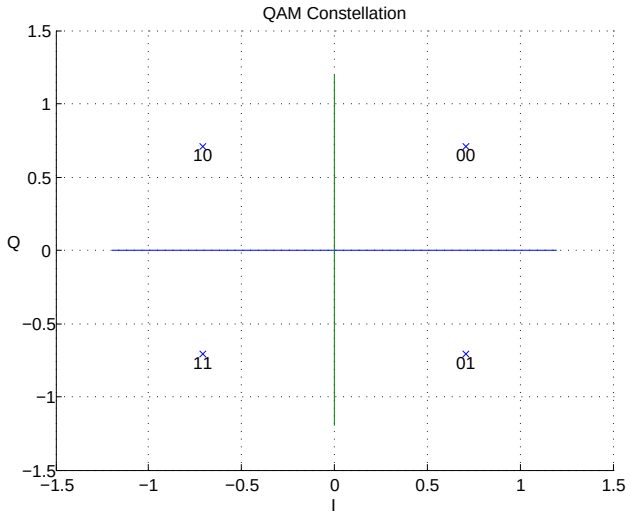


Figure B.3: QAM constellation as defined in the LTE.

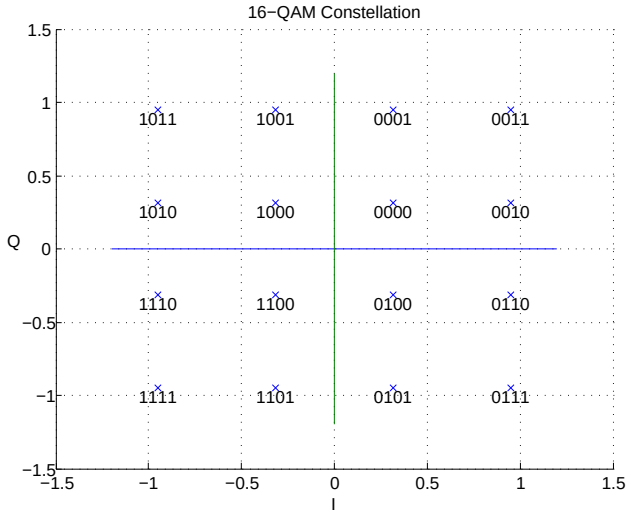


Figure B.4: 16-QAM constellation as defined in the LTE.

As seen, the QAM demodulation is straight forward to demodulate, however; with a higher alphabet size comes a greater challenge.

B.3 M-ary QAM

The 16-QAM and 64 QAM constellation are seen in figure B.4 and B.5

As with the QAM constellation, it is possible to split the decoding process up in small steps[16], which calculates the value of each bit seperatly, due to

[2]Error Correcting Codes
[3]Log-Likelyhood Ratio

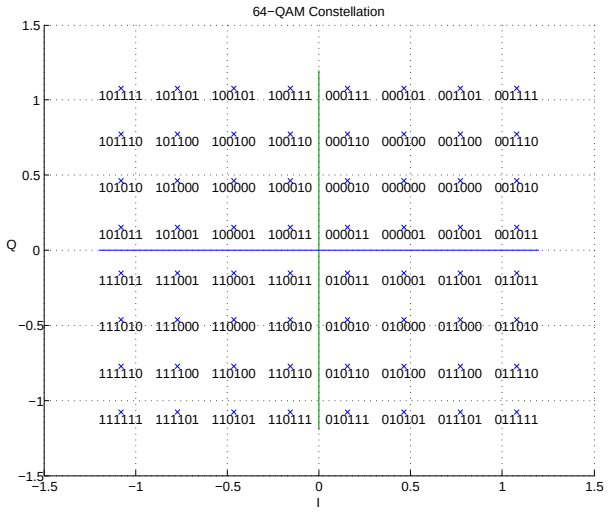


Figure B.5: 64-QAM constellation as defined in the LTE.

the Grey encoding.

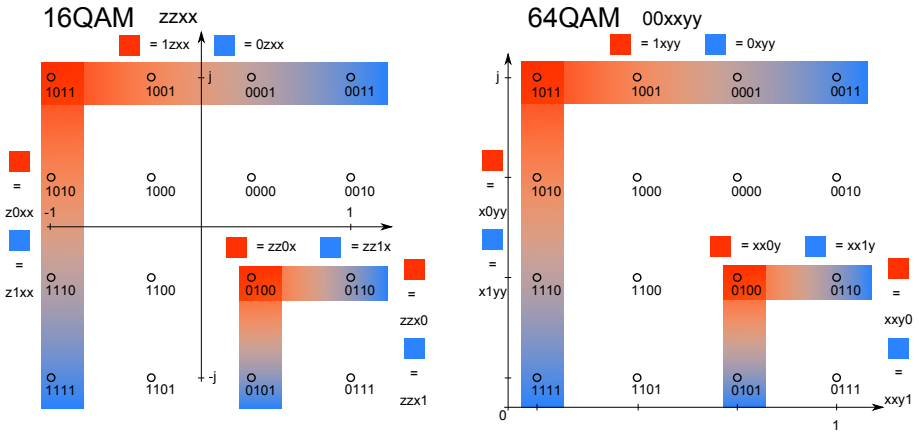


Figure B.6: This illustrates the system which the grey encoding creates. As can be seen the bits change in a pattern, that allows the ML decision rule to be used.

Figure B.6 illustrates the different regions and how they are related to the in-phase and quadrature components of the received symbol. Based on the deduction in [16] this illustration leads to the 16-QAM soft-output decoder algorithm displayed in equation B.11 to B.12, using the deducted likelihood ratio equation B.7.

$$D_{I,1} = -\frac{(\tilde{s}_{\text{real}}(k) - \frac{2}{\sqrt{10}})^2}{2 \cdot \sigma_n^2} + \frac{(\tilde{s}_{\text{imag}}(k) + \frac{2}{\sqrt{10}})^2}{2 \cdot \sigma_n^2} \tag{B.11}$$

$$D_{I,2} = \begin{cases} -\frac{(\tilde{s}_{\text{real}}(k) - \frac{1}{\sqrt{10}})^2}{2 \cdot \sigma_n^2} + \frac{(\tilde{s}_{\text{imag}}(k) + \frac{3}{\sqrt{10}})^2}{2 \cdot \sigma_n^2}, & \tilde{s} < -\frac{1}{\sqrt{10}} \\ \frac{\tilde{s}_{\text{real}}(k) - \frac{2}{\sqrt{10}})^2}{2 \cdot \sigma_n^2}, & |\tilde{s}| \leq \frac{1}{\sqrt{10}} \\ -\frac{(\tilde{s}_{\text{real}}(k) - \frac{3}{\sqrt{10}})^2}{2 \cdot \sigma_n^2} + \frac{(\tilde{s}_{\text{imag}}(k) + \frac{1}{\sqrt{10}})^2}{2 \cdot \sigma_n^2}, & \tilde{s} > \frac{1}{\sqrt{10}} \end{cases} \tag{B.12}$$

The imaginary part is calculated using the same equations replacing $\tilde{s}_{\text{real}}$ with $\tilde{s}_{\text{imag}}$. When the outputs has been calculated, the resulting bit sequence $\tilde{m}_s$ are to be interpreted as seen in equation B.13

$$\tilde{m}_s = [D_{I,1}, D_{Q,1}, D_{I,2}, D_{Q,2}] \tag{B.13}$$

In the same way, it is possible to deduct the 64-QAM soft-output decoder to be as seen in equation B.14 to B.16.

$$D_{I,1} = -\frac{(\tilde{s}_{\text{real}}(k) - \frac{4}{\sqrt{10}})^2}{2 \cdot \sigma_n^2} + \frac{(\tilde{s}_{\text{imag}}(k) + \frac{4}{\sqrt{10}})^2}{2 \cdot \sigma_n^2} \tag{B.14}$$

$$D_{I,2} = \begin{cases} \frac{\tilde{s}_{\text{real}}(k) - \frac{3}{\sqrt{10}})^2}{2 \cdot \sigma_n^2}, & |\tilde{s}| \leq \frac{3}{\sqrt{42}} \\ -\frac{(\tilde{s}_{\text{real}}(k) - \frac{3}{\sqrt{10}})^2}{2 \cdot \sigma_n^2} + \frac{(\tilde{s}_{\text{imag}}(k) + \frac{1}{\sqrt{10}})^2}{2 \cdot \sigma_n^2}, & |\tilde{s}| > \frac{3}{\sqrt{42}} \\ -\frac{(\tilde{s}_{\text{real}}(k) - \frac{1}{\sqrt{10}})^2}{2 \cdot \sigma_n^2} + \frac{(\tilde{s}_{\text{imag}}(k) + \frac{3}{\sqrt{10}})^2}{2 \cdot \sigma_n^2}, & |\tilde{s}| < -\frac{3}{\sqrt{42}} \end{cases} \tag{B.15}$$

$$D_{I,3} = \begin{cases} -\frac{(\tilde{s}_{\text{real}}(k) - \frac{3}{\sqrt{10}})^2}{2 \cdot \sigma_n^2} + \frac{(\tilde{s}_{\text{imag}}(k) + \frac{1}{\sqrt{10}})^2}{2 \cdot \sigma_n^2}, & |\tilde{s}| < -\frac{5}{\sqrt{42}} \\ -\frac{(\tilde{s}_{\text{real}}(k) - \frac{3}{\sqrt{10}})^2}{2 \cdot \sigma_n^2} + \frac{(\tilde{s}_{\text{imag}}(k) + \frac{1}{\sqrt{10}})^2}{2 \cdot \sigma_n^2}, & -\frac{5}{\sqrt{42}} \leq |\tilde{s}| < -\frac{3}{\sqrt{42}} \\ -\frac{(\tilde{s}_{\text{real}}(k) - \frac{3}{\sqrt{10}})^2}{2 \cdot \sigma_n^2} + \frac{(\tilde{s}_{\text{imag}}(k) + \frac{1}{\sqrt{10}})^2}{2 \cdot \sigma_n^2}, & -\frac{3}{\sqrt{42}} \leq |\tilde{s}| < -\frac{1}{\sqrt{42}} \\ \frac{\tilde{s}_{\text{real}}(k) - \frac{3}{\sqrt{10}})^2}{2 \cdot \sigma_n^2}, & |\tilde{s}| \leq \frac{1}{\sqrt{42}} \\ -\frac{(\tilde{s}_{\text{real}}(k) - \frac{3}{\sqrt{10}})^2}{2 \cdot \sigma_n^2} + \frac{(\tilde{s}_{\text{imag}}(k) + \frac{1}{\sqrt{10}})^2}{2 \cdot \sigma_n^2}, & \frac{1}{\sqrt{42}} < |\tilde{s}| \leq \frac{3}{\sqrt{42}} \\ -\frac{(\tilde{s}_{\text{real}}(k) - \frac{3}{\sqrt{10}})^2}{2 \cdot \sigma_n^2} + \frac{(\tilde{s}_{\text{imag}}(k) + \frac{1}{\sqrt{10}})^2}{2 \cdot \sigma_n^2}, & \frac{3}{\sqrt{42}} < |\tilde{s}| \leq \frac{5}{\sqrt{42}} \\ -\frac{(\tilde{s}_{\text{real}}(k) - \frac{3}{\sqrt{10}})^2}{2 \cdot \sigma_n^2} + \frac{(\tilde{s}_{\text{imag}}(k) + \frac{1}{\sqrt{10}})^2}{2 \cdot \sigma_n^2}, & \frac{5}{\sqrt{42}} < |\tilde{s}| \end{cases} \tag{B.16}$$

Similar to the 16-QAM algorithm, the bits decided by the imaginary part, can be calculated by replacing $\tilde{s}_{\text{real}}$ with $\tilde{s}_{\text{imag}}$ and then the multiplexing should be as in equation B.17

$$\tilde{m}_s = [D_{I,1}, D_{Q,1}, D_{I,2}, D_{Q,2}, D_{I,3}, D_{Q,3}] \tag{B.17}$$

Appendix C

The initial arm stretches when making USRP2 firmware

This appendix serves as a documentation on the initial processes, before the new FPGA image has been designed, written and tested. This Appendix is made due to lack of gathered documentation on how to use/compile/reprogram the USRP2, and will therefore ensure that anyone are able to repeat the tests that has been done in this report. The different documentation pages used are also stored on the CD under the folder "diff_documentation" to ensure availability.

C.1 Preparing to work with the USRP2

These are the initial steps which has to be done in order to begin using the USRP2[1]:

1. Clone and build the UHD as described in the build manual:

   http://files.ettus.com/uhd_docs/manual/html/build.html[2]

2. Install Xilinx ISE Design suite[3] (note that this is the paid version), following the guide given on:

   - *http://ubuntuforums.org/showthread.php?t=1547435* [4]

3. If you dont want to compile the firmware of the USRP2, download it from Ettus' homepage

   - *http://files.ettus.com/uhd_releases/master_images/*

4. Build the USRP2 FPGA image

[1] Experience has shown that a Linux partition with at least 30 Gb is required, in this project Ubuntu was installed on top on Windows using the *Wubi* installer.

[2] Found as diff_documentation/uhd_build_manual

[3] At the current time versions newer than 12.2 would not compile with met timing constraints as of UHD version

[4] found as diff_documentation/xilinx_build_manual

- *cd <uhd path>/fpga/usrp2/top/usrp2; make bin.*

5. If the compilation results in error try exporting the following libraries:

   - *export LD_PRELOAD=$LD_PRELOAD: /opt/Xilinx/12.1/ISE_DS/ISE/lib/lin64/libAntlr.so: /opt/Xilinx/12.1/ISE_DS/ISE/lib/lin64/libstlport.so.5.1*

6. Burn the FPGA image and firmware onto the SD card using the python gui supplied with the UHD:

   sudo python <uhd path>/host/utils/usrp2_card_burner.py

   fpga image is located at */<uhd path>/fpga/usrp2/top/USRP2/build*

   fpga firmware is located whereever it was stored in step 3

If the USRP2 two seems to load the binaries from the SD card correctly, test that the FPGA image works properly by running one of the build examples supplied with the UHD build[5].

C.2 Implementing on the USRP2 in correspondence with the Vita Protocol

According to the report which has analysed the USRP2 FPGA image structure[33], a good place to add extra data processing modules are between the *VITA_framer* and *dsp_core* module, due to the fact that it is easy intercept the samples here, as illustrated on Figure C.1. The first thing that will be presented is a block diagram which describes the different inputs and outputs of the modules in the USRP2, which are close to the chosen point of implementation. Secondly, the dataflow of the signals will be analysed. Both the connections and signal dataflow has been deducted through the development of this report.

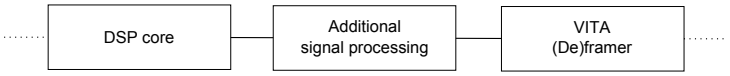


Figure C.1: The place of implementation for extra dataprocessing modules in the USRP2's firmware.

To deduce how the data interconnections and dataflow of the USRP2 FPGA image works, different steps has been taken.

C.2.1 Step 1 - A walk through the USRP2 FPGA image code

Before being able to add new modules to the USRP2 FPGA image, it is important to understand the signal flow at the point of interest in the code supplied by ETTUS research. In this case the point of interest are the in/outputs of the *dsp_core* and *vita_chain*, since it is in between these the data processing

[5] */<uhd path>/host/build/examples*

module should be connected.

The analysis has been conducted with the means of the *vita_chain_rx* testbed[6] supplied by ETTUS and a thorough analysing of the supplied Verilog code. The results are presented in Figure C.2 and Table C.1, containting the data variable names.

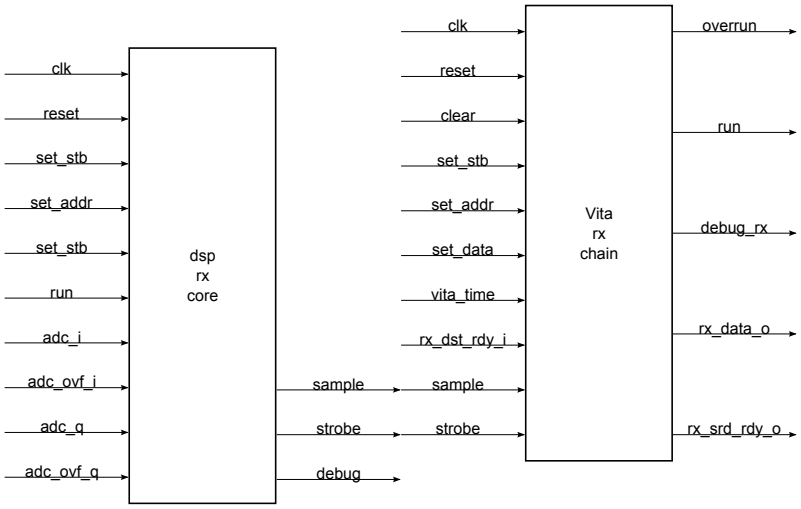


Figure C.2: The inputs, outputs and connections of the *dsp_core* and *vita_chain* module. This has been deduced by analysing the code located in */<uhd path>/fpga/usrp2/vrt*.

Variable name	Width	Functionality
clk	1	The master clock of the USRP2, which has the clock frequenzy of 100 MHz.
reset	1	Master reset which resets (initilializes) the USRP2 after the USRP2 firmware image has been loaded. Checket by the *sysctrl* module.
clear	1	Used to reset the data processing modules when a new transmission/reception are to commence. Set by the setting registers since these receive commands from the host-pc.
set_stb	1	Strobe signal for the setting registers. On a positive transition the setting register chosen by *set_addr* takes the current value of *set_data*. The strobe are controlled by by the ZPU.

Continued on next page

[6] <uhd-repo-path>/fpga/usrp2/vrt/vita_chain_rx_tb

Table C.1 – continued from previous page

Variable name	Width	Functionality
set_addr	8	Used to choose the desired setting registers. The adress are controlled by by the ZPU, and determined by the host-PC.
set_data	32	Holds the data which should be written to the given setting register. The data are given by the host-PC through the ethernet port.
vita_time	{32, 32}	Holds the current internal time, used to tag each incoming/outgoing sample or to handle delayed transmission. The time is defined in {seconds, ticks*rbrace* where "ticks" has the resolution of $\frac{1}{512}$ s.
sample	{16, 16}	The recieved samples from the ADC's. The actual ADC data has been rounded to an precision of 16 bits and are held in the variable *sample* as {in-phase sample, quadrature sample}.
strobe	1	The sample output/input of/to the *dsp_rx_chain*/*dsp_tx_chain* comes at a rate defined by a decimated version of the *clk* signal. *strobe* are the clock signal which comply with this decimated data rate.
rx_dst_rdy_i	1	Sets if the packet router is ready to receive samples.
run	1	Signal set by the *vita_rx_chain*/*vita_tx_chain*, which enables the *dsp_rx_core*/*dsp_tx_core*.
adc_i & *adc_q*	24	the in-phase and quadrature sample from the ADC's.
adc_ovf_i & *adc_ovf_q*	1	Signals if the *dsp_rx_core* has been to slow to gather the samples, resulting in overflow of the adc's buffer.
debug	32	All of the debug outputs seen in the system can be wired to the 32 bit wide debug port avaliable on the USRP2's board. This port can then be used with an digital analyser to debug the signal flow.

Table C.1: The different variables which has been analysed in the code and what these are used for.

C.2.2 Step 2 - Defining the test environment

Since this appendix serves as the documentation on how the development process has proceeded, the chosen testbed environments will be described here.

To verify the research analysed in this chapter three testbeds has been created, which will be described in the next sections.

C.2.2.1 C++ program, which prints the output samples from the USRP2 to the terminal.

This testbed has the functionality of displaying the outputs from the USRP2 directly to the terminal. It is designed with the purpose of testing static input data in mind.

Static input data is used in the initial tests and in the test of the different data processing units, such as the FFT. The static input data will only last for a certain amount of packets, ergo it is acceptable to analyse the results i the terminal.

The testbed is written in C++, is named *USRP_firmware_check* and has three program options:

–*sps*: Sample rate in samples per second.

–*spp*: Samples per packet, defines the width of the buffer that are to be used to receive data from the USRP2.

–*nop*: Number of packets which defines how many packets should be received.

It is to be called from a Linux terminal, whereafter it initializes the connected USRP2 and then displays a number of packets, defined by –nop, with length defined by –spp. The output are displayed in the terminal and each sample are given a number.

C.2.2.2 Verilog HDL simulator (Isim)

The verilog simulator serves the purpose of analysing the timing and signal interaction of the developed modules.

To analyse the verilog code, the Xilinx provided simulator ISIM is used. ISIM is integrated in the ISE design suite and works by compiling the Verilog/HDL code into a runable C++ program. This can be used to analyse the in/outputs of the modules and single step through the process, simulating the hardware calculation process.

By creating a testbed in Verilog which mimics the signal flow of the place where the module is to be implemented, it can be checked if the designed module operates as desired. The testbed was created to simulate the interaction with the different signals connecting between the *dsp_core_rx* and *vita_rx_chain* modules as on the USRP2.

The testbed tests the included modules for their compatibility to the attached modules. It is desired that the newly designed modules follows the

current dataflow of the USRP2, since this would make it transparent to the system. In this case transparency is desired due to the fact that it will ease the implementation of the module since no other modules has to be re-written.

The signals of interest are illustrated in Figure C.2 and Table C.1 and their dataflow are illustrated in Figure C.3. From the figure the interaction and the timing of the signals can be seen.

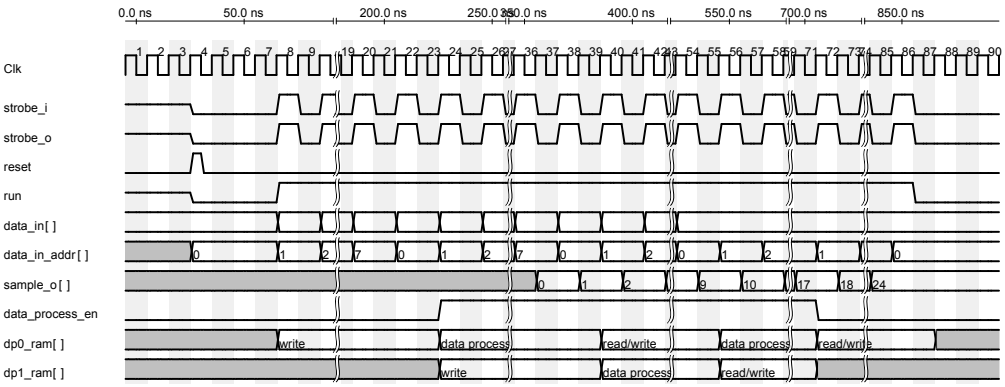


Figure C.3: This table shows the interaction and timing of the signals of the developed testbed.

C.2.2.3 Precompiled example *dft_ascii_art* provided by ETTUS research in the UHD - example folder.

The Precompiled example is used to check the compatibility of the designed modules and the current USRP2 functions. This precompiled example gathers samples from the USRP and makes a FFT, of which the result is shown in the terminal as ASCII-art. If this example is able to work perfectly with the new designed modules, there is a good chance that the USRP2 still works as initially.

With the three testbeds described, it can be verified if the dataflow, timing and signal interaction is as desired and that the program is compatible with the existing USRP2 FPGA image.

C.2.3 Step 3 - finding the right input

The first step is taken to see how it is possible to intercept the samples and manipulate them before they are sent to the pc. This step also serves to analyse the format of the input used.

As can be seen on figure C.2 one of the inputs to the *vita_rx_chain* module is *sample*, which is where the output from the dsp arrives. Through analysis of the *dsp_core_rx* module, it was deducted that the data variable *sample* was 32 bit wide and held the data as:

- sample[31:16] = 2's complement real part of the received sample.

- sample[15:0] = 2's complement imaginary part of the received sample.

The first part of step 1 is to unwire the *sample* input of the module and replace it with a constant value of 1 which is done as seen in the listing below:

```
579   reg [15:0] real_value = 16'b0100000000000000;
580   reg [15:0] imag_value = 16'b1110000000000000;
581   wire strobe_our0 = strobe_rx0;
582   wire[31:0] sample_our0 = {real_value, imag_value};
583   wire[31:0] sample_our1 = {real_value, imag_value};
584
585    vita_rx_chain #(.BASE(SR_RX_CTRL0),.UNIT(0),.FIFOSIZE(DSP_RX_FIFOSIZE))
           vita_rx_chain0
586      (.clk(dsp_clk), .reset(dsp_rst), .clear(clear_rx0),
587       .set_stb(set_stb_dsp),.set_addr(set_addr_dsp),.set_data(set_data_dsp),
588       .vita_time(vita_time), .overrun(overrun0),
589       .sample(sample_our0), .run(run_rx0), .strobe(strobe_our0),
590       .rx_data_o(wr1_dat), .rx_src_rdy_o(wr1_ready_i), .rx_dst_rdy_i(
              wr1_ready_o),
591       .debug() );
```

Listing C.1: This listings shows the constant value which is set to be the input to *vita_rx_chain*. The chosen value of the constant will give an initial overview of if the 32 bit wide variable "sample" holds data as expected

When the simple c++ test program described in section C are run, the output written in the terminal should comply with the chosen input value:

- sample[31:16] = 0100000000000000 → real value = 0.5.
- sample[15:0] = 1110000000000000 → imag value = -0.25.

The result of the test was that the output behaved as expected and hereby that it is possible to intercept the sample dataflow between the *dsp_core* and *vita_chain* when using few added lines of code. The next step is to analyse how to use other variables, such as clk, reset and stb, to manipulate with the input.

C.2.4 Step 4 - Faking data processing

This test has the purpose of testing if the different signals, which is to be utilized, behaves like deduced.

The test code is an expansion of the code in the beforehand Listing C.1, which uses a static input to test if the signal flow is as analysed through the code.

Listing C.2 show the code for this test of signal flow. The real value of *sample* are used to test the reset and the master clock. By letting the data being controlled by *dsp_clk* line 583 to 591 should be running at the master frequency of 100 MHz.

The first thing which is tested are the *clear_rx0* command, which should ensure that the real value will always start being 0 for each new transmission.

As explained in Table C.1, the *strobe_rx0* is a decimated version of the *dsp_clk*, meaning that it will be in the **HIGH** state over multiple *dsp_clk*s. This

```
600   signed reg [15:0] real_value = 16'b1111111111111111;
601   signed reg [15:0] imag_value = 16'b0000000000000000;
602   wire strobe_our0 = strobe_rx0;
603
604   always @ (posedge dsp_clk) begin
605     if (clear_rx0) begin
606       real_value = 16'b0000000000000000;
607     end else if (run_rx0 && strobe_rx0) begin
608       real_value <= real_value+1;
609     end else begin
610
611     end
612   end
613
614   always @ (posedge (strobe_rx0 || dsp_rst)) begin
615     if (run_rx0)
616       imag_value = 16'b1111111111111111;
617     else
618       imag_value = imag_value + 1;
619   end
620
621   wire[31:0] sample_our0 = {real_value, imag_value};
622   wire[31:0] sample_our1 = {real_value, imag_value};
623
624    vita_rx_chain #(.BASE(SR_RX_CTRL0),.UNIT(0),.FIFOSIZE(DSP_RX_FIFOSIZE))
          vita_rx_chain0
625      (.clk(dsp_clk), .reset(dsp_rst), .clear(clear_rx0),
626       .set_stb(set_stb_dsp),.set_addr(set_addr_dsp),.set_data(set_data_dsp),
627       .vita_time(vita_time), .overrun(overrun0),
628       .sample(sample_our0), .run(run_rx0), .strobe(strobe_our0),
629       .rx_data_o(wr1_dat), .rx_src_rdy_o(wr1_ready_i), .rx_dst_rdy_i(
              wr1_ready_o),
630       .debug() );
```

Listing C.2: Verilog implementation of the code which is designed to test the different signals of the system.

means that the real value should increase by

$$\frac{1}{2^{15}} \cdot \frac{N_{\text{decimation}}}{2}, \qquad [-]$$

for each new received sample. To see the difference between the rates, the imaginary part are only being counted up once for each *strobe_rx0* signal. Furthermore, the imaginary part of the sample are not reset other than the first time.

After the code was compiled, it was tested using the c++ testbed described in Appendix C, which prints packets of samples to the terminal. The program has been run with the options *–spp* 128, *–nop* 2 and *–sps* 200000. It was run two consequently times to test the reset signal and the results are seen in Table C.2 and C.3. The expected results were created using a MATLAB script, seen on the cd in "/MATLAB/fake_dp_test.m".

Run 1 - Package 1	
Sample nr	value (I, Q)
0	(0.000061035, -0.99997)
1	(0.000122070, -0.99994)
2	(0.000183110, -0.99991)
.	.
.	.
126	(0.007751500, -0.99612)
127	(0.007812500, -0.99609)

Run 1 - Package 2	
Sample nr	value (I, Q)
0	(0.007873500, -0.99606)
1	(0.007934600, -0.99603)
2	(0.007995600, -0.99600)
.	.
.	.
126	(0.015564000, -0.99222)
127	(0.015625000, -0.99219)

Table C.2: The terminal output of the first run of the C++ test program. These values are to be compared to the values of the MATLAB program on the cd "/MATLAB/fake_dp_test.m".

Run 2 - Package 1	
Sample nr	value (I, Q)
0	(0.000061035, -0.99216)
1	(0.000122070, -0.99213)
2	(0.000183110, -0.99210)
.	.
.	.
126	(0.007751500, -0.98831)
127	(0.007812500, -0.98828)

Run 2 - Package 2	
Sample nr	value (I, Q)
0	(0.007873500, -0.98825)
1	(0.007934600, -0.98822)
2	(0.007995600, -0.98819)
.	.
.	.
126	(0.015564000, -0.98441)
127	(0.015625000, -0.98438)

Table C.3: The terminal output of the second run of the C++ test program. These values are to be compared to the values of the MATLAB program on the cd "/MATLAB/fake_dp_test.m".

Comparing the output in the terminal to the MATLAB script, showed that the signals behaved as analysed. This means that the signal flow has been analysed correctly so that it is possible to now make a template for the data processing module.

C.2.5 Step 5 - Creating a module template

After the different tests has been done and the signals function and flow has been verified, a template for the *data_processing* module can be established. This template serves the purpose of determining a framework for the design of the data processing modules.

The module template for the receiver side is named *pipeline_rx* due to the fact that the top functionality of this is to control the data gathering and starting/stopping the data processing. The template is seen in Listing C.3. The reason for the chosen connections are showed in Table C.4.

```
1  module data_process
2    #(parameter MEM_WIDTH = 1280,
3      parameter setting_reg_BASE = 160,
4      parameter L2_PACKET_WIDTH = 7,
5      parameter SAMPLE_RES = 32)
6    (input clk,
7     input reset,
8     input clear,
9     input run,
10    input [31:0] sample,
11    input strobe_i,
12    input [L2_PACKET_WIDTH-1:0] samples_pr_packet,
13    input [L2_PACKET_WIDTH:0] desired_fft_length,
14    output wire [31:0] sample_o,
15    output wire strobe_o);
```

Listing C.3: The template of the module which has been created through the analysis in appendix C.

Variable name	Direction	Motivation
clk	input	The master clock is used as input since this can be used to control the data processing, which then can operate faster than the decimated *strobe_i*.
reset	input	The system should be reset when the USRP2 has been initialized.
clear	input	Makes it possible to reset the module on a new recieve/transmit command.
run	input	The *data_processing* module should only be active when the other dps functionalities are.
sample_i	input	This is the sample input from the *dsp_rx_core* module and it is used to ensure that the *data_processing* module gets the right input.

Continued on next page

Table C.4 – continued from previous page

Variable name	Direction	Motivation
strobe_i	input	Connected to the *strobe* signal of the *dsp_rx_core* since this signals to the *data_processing* module knows that data is avaliable.
set_data, *set_addr* & *set_stb*	input	Used to enable the *data_processing* module to grab the settings of the system; eg. the actual package length.
sample_o	output	The output data which are data processed.
strobe_o	output	The *data_processing* module should be able to strobe the data out.

Table C.4: Motivation for the choice of in and outputs used in the *data_processing* module.

In order implement the module in the ETTUS image compilation, a makefile has been created for the specific module so the compiler where to look for it. For the first module another makefile was used as a template and stripped down and altered so it included the module. This makefile can be seen i listings C.4.

```
1 GR950_SRCS = $(abspath $(addprefix $(BASE_DIR)/../../../../Documents/svn/
      usrp2_utils/verilog/, \
2 first_module.v \
3 pipeline_rx.v \
```

Listing C.4: Makefile created from template of another makefile to use for the created module.

This appendix has described how the firmware and drivers for the USRP2 has been found and installed, with purpose of others being able to recreate the process. Furthermore the signals around the chosen spot of implementation has been analysed, resulting in a module template which determines the different signal that should be used in further implementation.

Appendix D

Creating Modules

This appendix documents the design process of the *data_processing module* which has been designed through the project period with purpose of implementing new data processing capabilities to the USRP2 FPGA image. To deal with the complexity of this module, it has been split into sub modules.

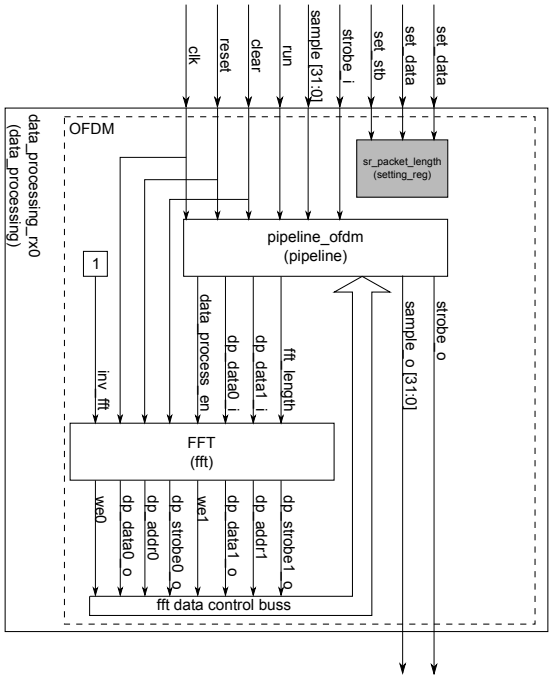


Figure D.1: How the modules are implemeted in the *data_processing* module and the interconnections between the modules. Each box represents a module where the name in parenthesis is the actual module name and the other is the name of the instanciated object. Note the grey box is a reused module designed by ETTUS research.

The sub modules of the *data_processing* module are described as they appear

in the hierarchy of the designed data processing unit, which is illustrated in Figure D.1. The actual implementational issues of the *data_processing* module is addressed in Section 5.3.

D.1 pipeline_rx

The data processing module *pipeline_rx* acts as a pipeline in the system, which in turn makes time for the data processing to take place. The pipeline structure allows the data processing to process the samples before they are transmitted to the *vita_rx_chain* module.

The basic idea of the memory structure is that while data is being loaded into one RAM block, the other RAM blocks data can be processed. In order to maintain the system dataflow, a special memory structure is needed. This can be achieved by using two memory registers and arrange them as is illustrated on Figure D.2. The different variables of the *pipeline_2n* module can be seen in Table D.1 and D.2, to ease the understanding of the code and the upcoming timing diagram.

Type	Name	Description
Parameter	MEM_WIDTH	Notes the width of the memory block.
	L2_PACKET_WIDTH	Notes the log2 of the packet width.
	SAMPLE_RES	Notes the sample resolution.
Input	clk	Inputs the clk from the dsp.
	reset	The reset signal if the system should reset.
	clear	The clear signal if the data should be cleared.
	run	Inputs the run enable.
	sample	Register holding the input samples.
	strobe_i	Input strobe.
	samples_pr_packet	Contains the number of samples per packet.
	desired_fft_length	The desired FFT length.
	dp_strobe	Data processing strobe.
	dp_addr	Data processing address.
	dp_data_i	Data processing data input.
Output	dp_data_o	Wire for the data processing output.
	data_process_en	Wire for the data processing enable.
	strobe_o	Wire for the output strobe.
	sample_o	Register holding the output samples.

Table D.1: Describes the input and outputs of the module pipeline.

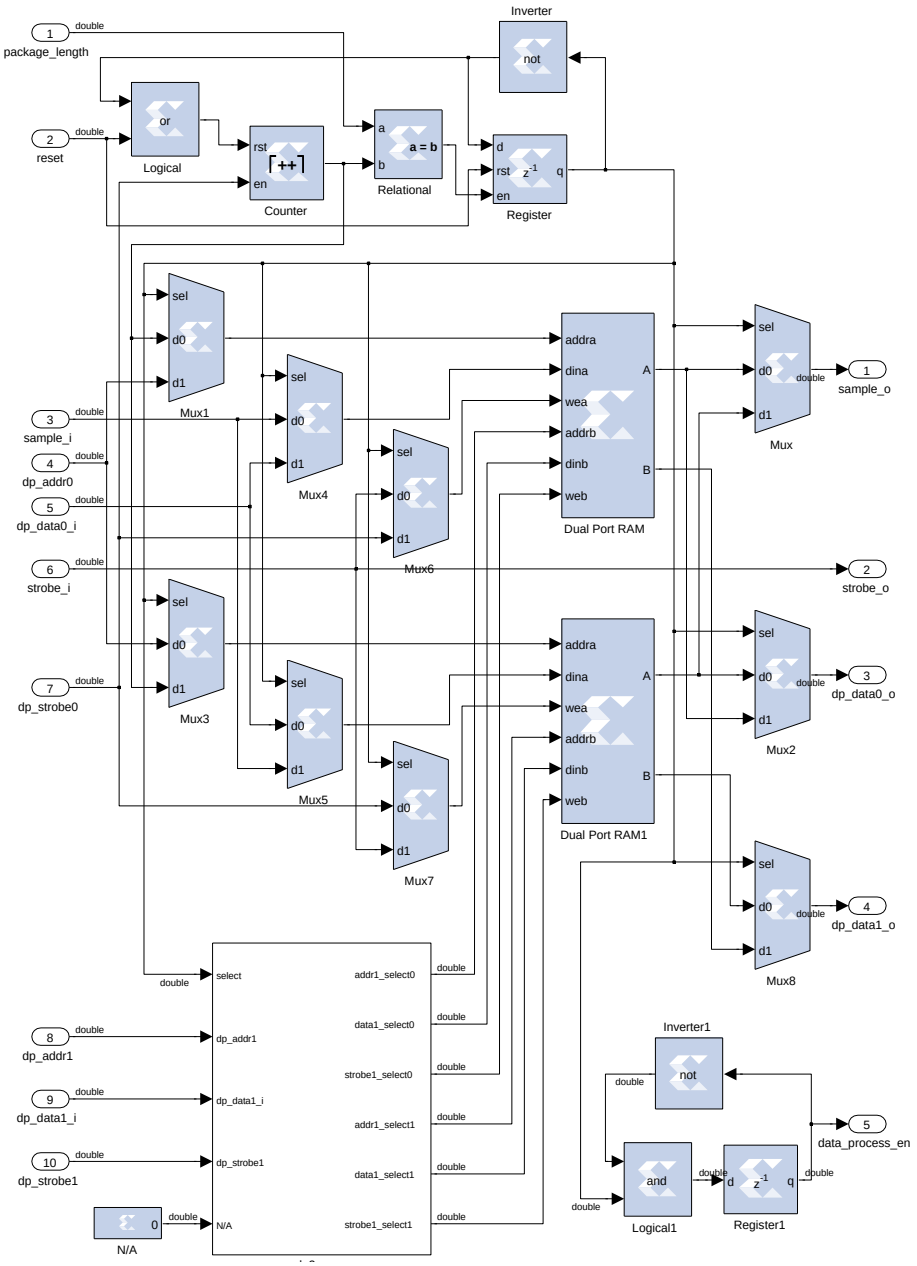


Figure D.2: System Generator block diagram over the pipeline module, showing the signal flow of the system.

Type	Name	Description
	pipe_setting_reg	Register holding the settings/states of the system.

reg Continued on next page

Table D.2 – continued from previous page

Type	Name	Description
	pkg_num	Register holding the package number. Helps define when data processing should start.
	current_count_state	Control signal that helps check for overflow
	prev_count_state	Control signal that helps check for overflow
	mem_count	A counter that counts up through the addresses
	dp0_ram_s0	Holds the strobe to the single access ram, for the two single access ram blocks.
wire	dp0_ram_addr0	Holds the address to the single access ram, for the two single access ram blocks.
	dp0_ram_data0_i	Holds the input data for the single access ram for when new data is to be written into the ram.
	dp0_ram_data0_o	Holds the output of the single access ram.
	strobe_i_or_reset	Initializes/resets the pipeline control signals, so it is ready for the next actions.

Table D.2: Describes the variables used in the module pipeline

To illustrate the functionality of the system, timing diagrams have been created to show the interconnection of the different functionalities. The initial timing idea is to delay *strobe_o* such that data will not clock out before the data is processed, eg. only after t_{delay} has passed. This timing is illustrated in Figure D.3.

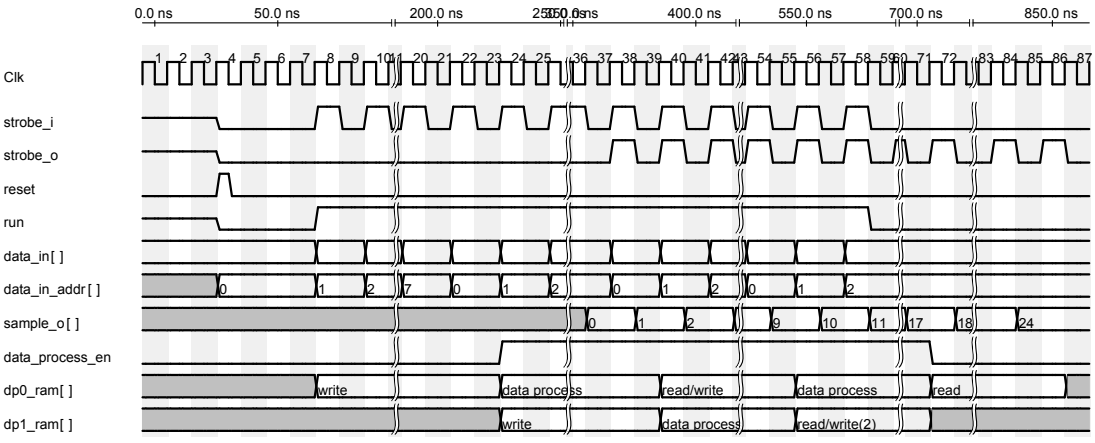


Figure D.3: Timing diagram of the first proposed timing diagram for the pipeline functionality, where the data is not clocked out until after t_{delay} has passed and the first data packet has been processed,

D.1.1 Test

Initial tests, showed that the timing described in Figure D.3 was not compatible with the existing Host-PC USRP communication, due to timing problems. It was quickly deduced that this could only be fixed by rewriting some of the code supplied by ETTUS, however this was not desired since it would mean complicating the implementation of the module. To ease the implementation another timing principle was proposed, as seen on Figure D.4.

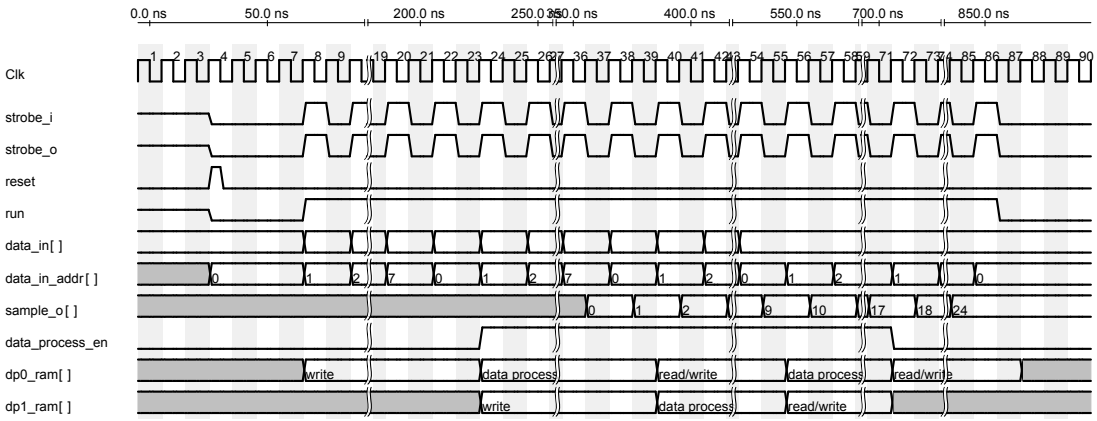


Figure D.4: The second timing proposal where *strobe_o* only has an added propagation delay. The disadvantage is that the first packets, corresponding to N_{packets}, does not contain any valid information, hence an extra control signal has to be set to tell when data is valid.

A disadvantage of this choice of timing is that it means that the data streaming will not be totally transparent to the user, since the first packets to be received are invalid due to the fact that they will contain NULLs. However, this problem is to be fixed within other parts of the framework.

D.1.1.1 Code

Here, some of the key features of the code are described, such that the reader will is to gain an overview of the basic ideas.

The dual access RAM blocks seen on Figure D.2 are a module created by ETTUS research called *ram_2port* which has been reused in this module to ensure compatibility with the compiler settings. When the enable input for one of the data access input, either *ena* or *enb*, are set, the corresponding output will take value of the memory on the current address on the input clock; *clka* or *clkb*. Furthermore, if write enable, either *wea* or *web*, are set, the ram block on the address will take value of the input; *dia* or *dib* on a clock strobe. For each stage n of the pipeline, a ram block is added.

Besides the ram block(s), there are two key functionalities of this system. First of is the code which ensures that the first datablock will be collected

```
112   always @ (posedge clk) begin
113     if (pkg_num == 1 && run == 1 && (prev_mem_state != current_mem_state))
114       pkg_num = 2;
115     else if (pkg_num != 1 && run == 1)
116       pkg_num = 0;
117     else
118       pkg_num = 1;
119   end
```

Listing D.1: The part of the code which checks if the first packet of the data reception has been collected, which would imply that data processing can commence.

```
144   wire strobe_i_or_reset = strobe_i || reset || clear;
145   always @ (posedge strobe_i_or_reset) begin
146     if (~run) begin
147       mem_count <= 0;
148       prev_mem_state <=0;
149       data_process_en <= 0;
150       sample_o <= 0;
151
152     end else begin
153       mem_count <= mem_count + 1;
154       #1 current_mem_state <= (mem_count == samples_pr_packet) ^
              prev_mem_state;
155       #1 sample_o <= current_mem_state ? dp1_ram_data0_o : dp0_ram_data0_o;
156       #1 prev_mem_state <= current_mem_state;
157       #1 data_process_en <= (current_mem_state != prev_mem_state) && pkg_num
              != 1;
158     end
159   end
```

Listing D.2: The *pipeline_n2* code which controls the data storing process.

before processing starts. This code is seen in Listing D.1.

As seen in Listing D.1, if *run* is not high, the packet number is set to 1, which illustrates that this is the first incoming packet of a new reception.

The variable *current_mem_state* increases with one each time one ram block has been filled, which makes it useful to control the multiplexers to the memory registers. Furthermore, as seen in Listing D.1 it is used to see if the first stage of the pipeline has been filled, which tells if the data processing can commence.

The other important function of the *pipeline_n2* module is the data storage control. This ensures that the data are stored in the correct order in the RAM block and sets the *data_process_en* at the correct time, as can be seen on listings D.2.

As in the previous part, the system resets itself whenever it gains a *reset* or *clear* signal and *run* is not set. When run is high and the module gets a *strobe_i* it goes directly into the desired *ram* module which ensures that the data is stored at the correct address. When the data is stored, the memory address increases by one, whereafter the system checks if the memory registers has been filled up. If the given memory register is filled, it will select the next memory register, using the muxes, and ensure that the data processing unit can get access to the data to be processed.

D.1.2 Test

Different tests has been made on this system. One of the tests was done by writing a verilog testbench and use it to debug the code using the Xilinx ISim GUI as the one described in Appendix C. Another test was done by using the precompiled C++ example *dft_ascii_art* to check if the samples were still gathered.

The purpose of the Verilog testbench was to check if the code stored the data in the correct RAM blocks in the correct order. The testbench can be found on the attached CD as *pipeline_n2_tb* in the *verilog* folder. It was shown that the data was stored correctly, since the fictive sample inputs were stored in the RAMs as expected.

The test using the *dft_ascii_art* had the purpose of testing if the pipeline module worked coherently with the supplied USRP2 FPGA code. The test was done by implementing the *pipeline_n2* module directly into the *u2_core* module of the ETTUS FPGA image. The module was placed in between the *dsp_core* and vita_rx_chain modules, as seen in Listing D.3.

As seen in Listing D.3 the interconnections for the data processing are not connected, which is because of the fact that they are not used in the current test.

If the *dft_ascii_art* example, also described as a testbed in Appendix C, works as before the module implementation, an initial conclusion can be made that the pipeline module is transparent for the communication between the UHD driver and FPGA image. The test setup is made as on Figure D.5.

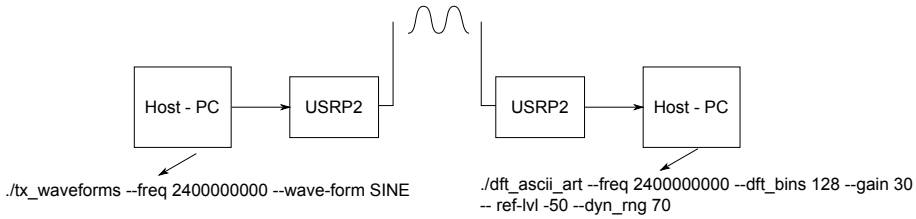


Figure D.5: The test setup to test if the implementation of the *pipeline_2n* module still supports the original UHD examples. The commands under the host pc's are the commands used in the Linux terminal on the given PC.

The results of the tests was that the the *dft_ascii_art* ran as expected since the frequency of the sine transmitted from the tx_host was displayed in the ascii art in the terminal.

D.2 FFT module

As explained in Section 2.4.2, the base of the OFDM modulation is the (I)FFT which transfers the subcarriers between the frequency and time do-

```
567     dsp_core_rx #(.BASE(SR_RX_DSP0)) dsp_core_rx0
568       (.clk(dsp_clk),.rst(dsp_rst),
569        .set_stb(set_stb_dsp),.set_addr(set_addr_dsp),.set_data(set_data_dsp),
570        .adc_i(adc_i),.adc_ovf_i(adc_ovf_a),.adc_q(adc_q),.adc_ovf_q(adc_ovf_b)
               ,
571        .sample(sample_rx0), .run(run_rx0_d1), .strobe(strobe_rx0),
572        .debug() );
573
574     setting_reg #(.my_addr(SR_RX_CTRL0+3)) sr_clear_rx0
575       (.clk(dsp_clk),.rst(dsp_rst),
576        .strobe(set_stb_dsp),.addr(set_addr_dsp),.in(set_data_dsp),
577        .out(),.changed(clear_rx0));
578
579   wire [31:0] pipeline_sample_o;
580   wire pipeline_stb_o;
581
582   pipeline_n2 #(.MEM_WIDTH(128), .setting_reg_BASE(160), .L2_PACKET_WIDTH(7),
           .SAMPLE_RES(32)) pipeline_n2_0
583   (.clk(dsp_clk),
584    .reset(dsp_reset),
585    .clear(clear_rx0),
586    .run(run_rx0),
587    .sample(sample_rx0),
588    .strobe_i(strobe_rx0),
589    .samples_pr_packet(7'b1111111),
590    .dp_strobe(), .dp_addr(), .dp_data_i(), .dp_data_o(), .data_process_en(),
591    .sample_o(pipeline_sample_o),
592    .strobe_o(pipeline_stb_o));
593
594    vita_rx_chain #(.BASE(SR_RX_CTRL0),.UNIT(0),.FIFOSIZE(DSP_RX_FIFOSIZE))
           vita_rx_chain0
595      (.clk(dsp_clk), .reset(dsp_rst), .clear(clear_rx0),
596       .set_stb(set_stb_dsp),.set_addr(set_addr_dsp),.set_data(set_data_dsp),
597       .vita_time(vita_time), .overrun(overrun0),
598       .sample(pipeline_sample_o), .run(run_rx0), .strobe(pipeline_stb_o),
599       .rx_data_o(wr1_dat), .rx_src_rdy_o(wr1_ready_i), .rx_dst_rdy_i(
              wr1_ready_o),
600       .debug() );
```

Listing D.3: Implementation of the *pipeline_n2* code in the prewritten ETTUS FPGA image. Note that the input to the *vita_rx_chain*, which was originally *sample_rx0* and *strobe_rx0* are instead inputs to the *pipeline_2n*. The inputs to the *vita_rx_chain* module are replaced by the *pipeline_n2* outputs *pipeline_sample_o* and *pipeline_stb_o*.

main. Firstly the (I)FFT is be analysed with respect to the coherency between dataflow, resource usage and timing constraints, whereafter the key parts of the design are explained.

This design uses the Cooley Tukey algorithm for the FFT. This report will not go into detail on how the Cooley Turkey algorithm works, but a quick walk-through on the important subjects of this algorithm are seen in Section 4.5.3.1.

D.2.1 Module timing analysis

The timing constraints for the FFT is set by the symbol time defined in the LTE standard, which is 71.42 μs including the cuclic prefix. However, it is possible to increase the available time for data processing, at the cost of resources and latency, by insertion of a pipeline.

Table D.3 states some of the timing and system constraints when implementing different pipeline sizes for an given FFT length, which means that it is worst case scenario for the system concerning timing. The memory usage in bits is calculated by Equation D.1.

$$mem_{\text{use}} = (N_{\text{FFT}} + N_{\text{CP}}) \cdot N_{\text{pipeline}} \cdot sample_{\text{resolution}} \quad \text{[b]} \quad \text{(D.1)}$$

The number of butterflies used to calculate a given FFT using this algorithm can be seen on Figure 4.7 in Section 4.5.3.1 to be:

$$N_{\text{butterflies}} = \frac{N_{\text{FFT}}}{2} \cdot \log 2 \left(N_{\text{FFT}} \right) \tag{D.2}$$

$N_{pipeline}$	2		3		4	
N_{FFT}	1024	2048	1024	2048	1024	2048
Memory usage [kb]	139.5		209.3		279	
FFT ressource usage	FFT_{res}		$2 \cdot FFT_{res}$		$3 \cdot FFT_{res}$	
Latency [μs]	142.8		214.26		357	
$^{clk}/_{sym}$	7142		14284		21426	
$^{clk}/_{samp}$	3.48		6.97		10.46	
$^{clk}/_{but}$	1.39	0.63	2.78	1.268	4.18	1.9

Table D.3: The different constraints and corresponding resource usage for implementation of different number of pipeline stages. FFT_{res} are the resources used for a single FFT, which are now known at the current state. The numbers are rounded down and is based on the 100 MHz clock on the USRP2 which feeds the FPGA.

Table D.4 contains an estimation of the resources and clock cycles per butterfly $^{clk}/_{but}$ for a certain amount of instant butterfly operations. This table has been utilizes to gain an overview of what kind of system should be designed to fulfil the timing requirements. The most important information are the $^{clk}/_{but}$ variable which indicates whether it is realistic to implement the given functionality. If the $^{clk}/_{but}$ in Table D.4 is lower than the one in Table D.3 it means that implementation of the given flow is possible.

N_{mem}	$clk_{load+store}$ S	$clk_{load+store}$ D	parallel butterflies	$^{clk}/_{but}$ S	$^{clk}/_{but}$ D	$N_{multipliers}$
1	2	2	0	∞	∞	0
2	4	2	1	4	2	4
3	6	4	1	6	4	4
4	8	4	4	2	1	16
5	10	6	4	2.5	1.5	16
6	12	6	5	2.4	1.2	20
7	14	8	5	2.8	1.6	20
8	16	8	12	1.34	0.66	48
9	18	10	12	1.5	0.84	48
10	20	10	13	1.54	0.76	52
11	22	12	13	1.7	0.92	52
12	24	12	16	1.5	0.76	64
13	26	14	16	1.62	0.88	64
14	28	14	17	1.64	0.84	68
15	30	16	17	1.76	0.94	68
16	32	16	32	1	0.5	128

Table D.4: An estimate of the resources required to obtain a given $^{clk}/_{but}$ rate. N_{mem} are the number of loaded memory registers and clk_{load} are the number of clock cycles needed to load them with single (S) or dual (D) access ram. Parallel butterflies are the number of butterflies able to run simultaneously, based on the amount of memory registers loaded. The $N_{multipliers}$ are an estimate of the multipliers needed if all the butterflies were run in parallel, without any optimization.

By comparing the intermediate results in Table D.3 and D.4 it is seen that there should be up to 8 parallel butterfly operations before the timing constraints are close to be met. By this it is clear that the timing constraints of a 2048 point FFT sets high requirements compared to the available resources and time in the system.

It has been chosen to implement a 1024 point FFT using dual port ram and four parallel butterflies, due to the fact that this is a very plausible solution since the constraint are at 1.39 $^{clk}/_{but}$ but the calculation can have a minimum of only 1 $^{clk}/_{but}$ according to Table D.3 and D.4. This means that this design will only support a bandwidth of up to 10 MHz. Based on the test results of this design, it will be clear if it is a plausible solution to implement the support of 20 MHz on the FFT.

Since the basic approach has been determined it is now possible to design the desired dataflow of the FPGA implementation. This will then be the guideline for the actual design of the implementation on the FPGA fabric.

D.2.2 4-point fft data flow

To illustrate the dataflow a dataflow graph is constructed, based on the butterfly structure in Figure 4.7 earlier in this section. Since this dataflow graph serves the purpose of aiding the design process, it is designed in such

a way that each algebraich- or memory operation which has the same vertical alignment are taken care of by the same part of hardware.

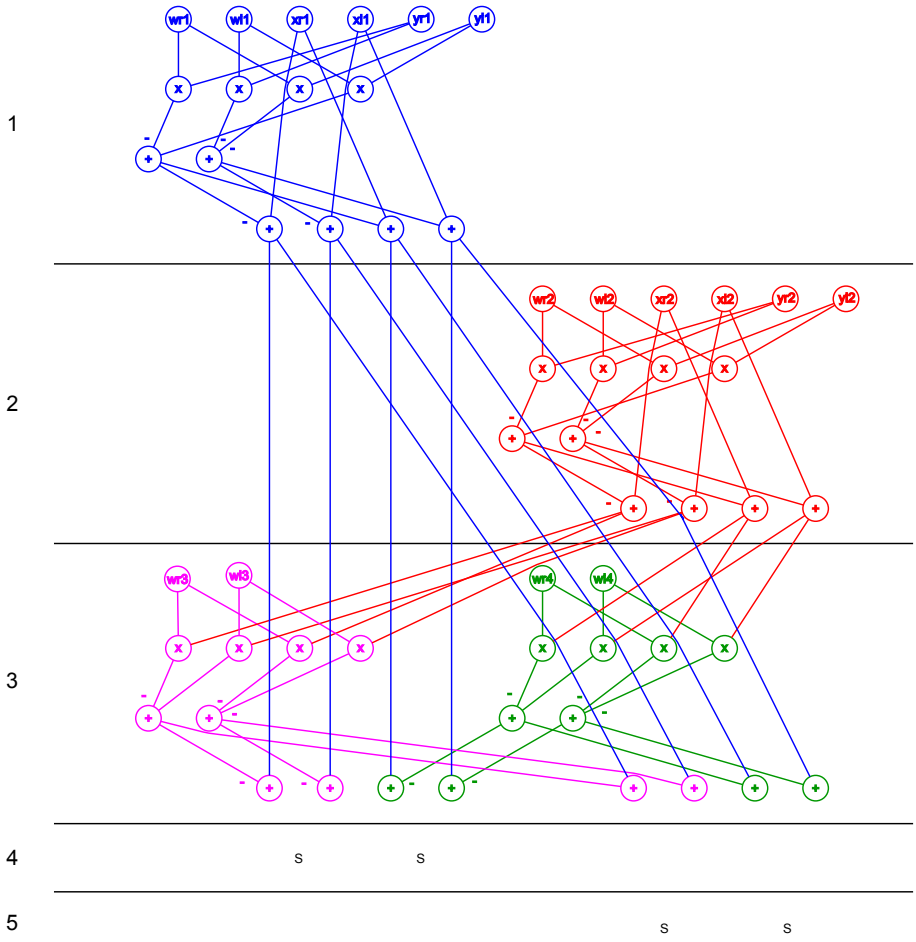


Figure D.6: The final dataflow of the 4-point (I)FFT used in this project. This graph has been created based on an analysis of Equation 4.24 to 4.27.

The dataflow graph is shown on Figure D.6. There are two key optimization steps which has been taken while developing this graph which and they are described briefly below.

The most apparent difference between Figure D.6 and the initial analysis in Table D.4 is that it only uses 8 multipliers instead of the 16. This optimization is possible since the intermediate results from the butterfly could be calculated while loading the data, which effectively reduces the amount of parallel butterflies by 2.

A second, less apparent, optimization is made in the actual butterfly operation, reducing it by 2 multiplications compared to the initial Equations 4.24 to 4.27. As seen in Equation D.3 to D.8 this has been done by calculating two intermediate results (y_{temp0} and y_{temp1}) and reusing them in the calculations of the output.

$$y_{temp0} = -w_r \cdot y_r + w_i \cdot y_i \tag{D.3}$$
$$y_{temp1} = -w_r \cdot y_i - w_i \cdot y_r \tag{D.4}$$
$$X_r = x_r - y_{temp0} \tag{D.5}$$
$$X_i = x_i - y_{temp1} \tag{D.6}$$
$$Y_r = x_r + y_{temp0} \tag{D.7}$$
$$Y_i = x_i + y_{temp1} \tag{D.8}$$

With this dataflowgraph it is now possible to design the general structure of the system.

D.2.3 4-point fft general architecture

This section covers the overall FFT design structure, meaning that it describes how the overall structure of the program is. By designing the overall structure before writing the code, it is possible to define, design and test different submodules independent of each other.

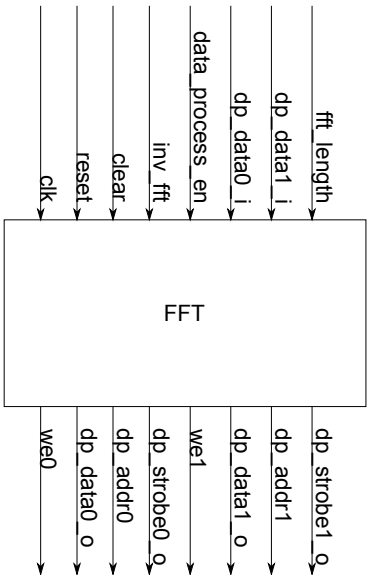


Figure D.7: In- and outputs of the FFT module. As seen the inputs consists of some control signals together with two data in ports which are used to read from the dual access ram in the pipeline. The outputs are pure ram access signals for reading and storing in the pipeline ram.

The first thing to define is the I/O's of the system, which is seen in Figure D.7. Together with the I/Os, the structure of the code is defined in Listing D.4.

```
25  ////////////////////////////////////////////////////////////////////
26  //                control signals                 //
27  ////////////////////////////////////////////////////////////////////
28  Internal registers which are used to control the basic flow of the program
        such as; loops
29
30  ////////////////////////////////////////////////////////////////////
31  //              intermediate value registers              //
32  ////////////////////////////////////////////////////////////////////
33  As seen on the dataflow, intermediate results has to be stored due to the
        fact that the different butterfly modules are reused
34
35  ////////////////////////////////////////////////////////////////////
36  //                  data_o wirering                 //
37  ////////////////////////////////////////////////////////////////////
38  The wirering that ensures that the ouput of the system (dp_data0_o and
        dp_data1_o) are correct. Here should be an implementation of the
        complex - real swap for the IFFT
39
40  ////////////////////////////////////////////////////////////////////
41  //            butterfly access control registers          //
42  ////////////////////////////////////////////////////////////////////
43  Control registers which holds the information of the systems current stage,
        used to calculate which data should be accessed and if the (I)FFT are
        done
44
45  ////////////////////////////////////////////////////////////////////
46  //       data access control, based on desired butterfly      //
47  ////////////////////////////////////////////////////////////////////
48  Calculation of the desired address access in the pipeline memory, based on
        the butterfly control registers
49  This data access control are created on the basis of the fact that the data
        are stored in the bit reversed order in the pipeline module
50   - Module name: data_addr_calc
51   - max propagation time: max half of a clock cycle
52
53  ////////////////////////////////////////////////////////////////////
54  //         twiddle address based on stage and buttefly nr       //
55  ////////////////////////////////////////////////////////////////////
56  Module which calculates the twiddle factor access based on the current
        state of the system held by the butterfly access control registers
57   - Module name: twitter_addr_calc
58   - max propagation time: max half of a clock cycle
59  ////////////////////////////////////////////////////////////////////
60  //              memory holding twiddle factors          //
61  ////////////////////////////////////////////////////////////////////
62  Rom module which holds the twiddle factors of which the output are
        controlled by the output of the twiddle address calculator
63   - Module name: twiddle rom
64   - max propagation time: max half of a clock cycle
65  ////////////////////////////////////////////////////////////////////
66  //         butterfly output based on the x and y input      //
67  //     wirering, strobe and if inputs are internal or external   //
68  ////////////////////////////////////////////////////////////////////
69  The wirering for the two butterfly modules which handles the actual
        calculation.
70
71  ////////////////////////////////////////////////////////////////////
72  //  butterfly output based on the x input and the intermediate y*w  //
73  //                    on strobe                   //
74  ////////////////////////////////////////////////////////////////////
75  Module which handles the butterfly calculation.
76   - Module name: butfly_calc
77   - max propagation time: max half of a clock cycle
78
79  ////////////////////////////////////////////////////////////////////
80  //         start calculation @ data_process_en pulse        //
81  //               and stop @ fft_done pulse              //
82  ////////////////////////////////////////////////////////////////////
83  Logic which starts/stops the FFT
84
85  ////////////////////////////////////////////////////////////////////
86  //   reset everything @ reset or end of calculation (fft_done)   //
87  ////////////////////////////////////////////////////////////////////
88  Logic which ensures that the FFT module is ready for its next calculation
89
90  ////////////////////////////////////////////////////////////////////
91  //         4 point fft using 2* butfly_calc         //
92  ////////////////////////////////////////////////////////////////////
```

```
93    Using the "clk" as a trigger this part will hold the signal flow of the
          system, which in the end controls the fft.
```

Listing D.4: The defined structure of the *fft* module.

The I/O's together with the code structure has been defined, it is possible to define the variables used in the main module of this design; *fft*. These definitions are seen in Table D.5 and will help when the design process is split into multiple processes.

Type	Name	Description
Reg	do_fft	Variable that if set to high, tells the system to do the fft calculations and if set to low tells the system that the fft is done.
	x_s2_b0	Intermediate values of the radix-4 butterfly calculation, where x denotes if it is the real or the imaginary value, "s" denotes which stage the butterfly calculation is at, b denotes which of the two radix-2 butterfly's being calculated.
	mem1_direct0	Control signal denoting if the given output pair of the fft should be from a register or directly from the calculation.
	input_strobe	Control signal which requests a change of address.
	stg_strobe	Control signal which requests a change in the stage for the twiddle factor calculation.
	get_new_but0	Control signal that requests for a new radix-2 butterfly calculation to start. Here "but" denotes which of the two radix-2 butterfly's should be used.
	ctrl_int1_ext0	Control signal that determines if the inputs for the radix-2 butterfly calculations should be internal intermediate or from a external register.
Wire	fft_done	Control signal that is set when the fft is complete.
	stage_count	Control signal denoting at what stage the fft calculation is.
	twiddle_stage	Control signal that is part of the twiddle factor address calculation.
	butterfly_input_count	Control signal denoting which input the radix-2 butterfly calculation should use at the given stage for the given pair.
	butterfly_pair_count	Control signal denoting what radix-2 butterfly pair is to be calculated at the given stage.
	data_addr0	Variables holding the data address, for the two addresses (address 0 and 1).

Continued on next page

Table D.5 – continued from previous page

Type	Name	Description
	data_addr0_bit_rev	Variables holding the bit reversed data address, for the two address (address 0 and 1).
	twiddle_addr0	Control signals denoting the addresses of the twiddle factor values.
	twiddle_wire_real0	Variables holding the real and imaginary value of the twiddle factor for both radix-2 butterflies.
	x0_temp_calc_in_r	Wires holding the real and imaginary inputs to the radix-2 butterflies x and y input.
	xr0_out	Wires holding the real and imaginary outputs of the radix-2 butterflies x and y output.
	X0	Wires which combine the real and imaginary outputs of the radix-2 butterflies x and y output.

Table D.5: The variables for the *fft* module.

When the desired submodules has been designed they should be implemented in the *fft* module and the signal flow should be establish and tested.

D.2.4 Submodules

This section will describe the different submodules which has been designed to the *fft* module. The modules will be described in the order they are written in the code, seen in Listing D.4 in the previous section.

D.2.4.1 Data access calculator module

This modules main function is to calculate the address which needs to be accessed in the *pipeline* module, based on the state of the (I)FFT. The module is named *data_addr_calc* and its I/Os are seen in Figure D.8.

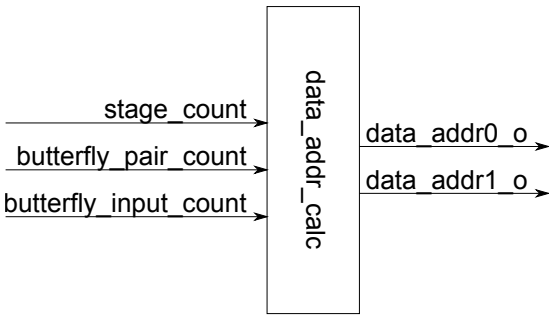


Figure D.8: In- and outputs of the *data_addr_calc* module, which handles the memory access addresses of the *fft* module.

The data address calculator is created based on the dual access ram scheme of the *pipeline* module, meaning that it will always output two addresses based on the input. The two addresses are determined to be to the x and y input of the butterfly calculation since this is what has been deducted through the creation of the dataflowgraph in Figure D.6 to be the desired way of implementation.

The address described by *data_addr0_o* defines the desired address for the X value seen in Equation D.3 to D.8 and *data_addr1_o* describes the address of the Y value. The addresses is calculated by the means of a logic circuit which uses all of the inputs. The address calculation is designed in such a way that it follows the pattern shown in Table D.6 which illustrates an 8-point FFT as in Figure 4.7.

stage	0			
butterfly_input_count	0	0	0	0
butterfly_pair_count	0	1	2	3
data_addr0_o	0	2	4	6
data_addr1_o	1	3	5	7

stage	1			
butterfly_input_count	0	1	0	1
butterfly_pair_count	0	0	1	1
data_addr0_o	0	1	4	5
data_addr1_o	2	3	6	7

stage	2			
butterfly_input_count	0	1	2	3
butterfly_pair_count	0	0	0	0
data_addr0_o	0	1	2	3
data_addr1_o	4	5	6	7

Table D.6: Illustration of the pattern which the *data_addr_calc* should mimic at the output.

The address control was tested by the use of a testbed, created to emulate the input signals to *data_addr_calc* from *fft*. The output signals *butterfly_input_count*, *butterfly_pair_count* and *stage_count* was then cross-referenced with Table D.6 and it was concluded that the module worked as intended.

D.2.4.2 Input and Pair Control Module

This module can be seen as a FSM which cycles through the control signals for the address calculation, the twiddle address calculation and the signal which terminates the fft, in the correct order. The module is named *input_pair_ctrl* and it's I/Os are depicted in Figure D.9.

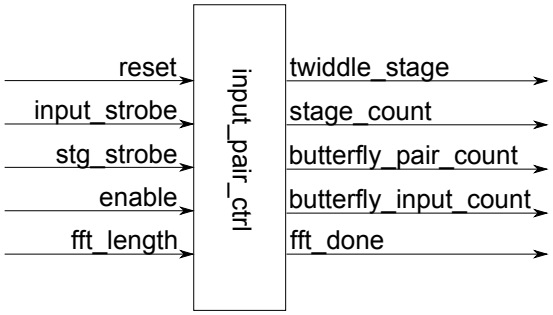


Figure D.9: In- and outputs of the *input_pair_ctrl* module, which creates most of the control signals for the *fft* calculation.

Each time one of the radix-2 butterflies are calculated, *input_strobe* is set to high in order for the control to ready the addresses of the next radix-2 butterfly calculation. The control signals *butterfly_input_count* and *butterfly_pair_count* are constructed using a counter (*some_counter*), the calculated maximum length of *butterfly_pair_count* and *butterfly_input_count* for the given *stage_count*, *butterfly_pair_max* and *butterfly_input_max* respectively. This is seen in Table D.7.

Number radix-4 butterfly calc	0				1			
Number radix-2 butterfly calc	0	1	2	3	4	5	6	7
butterfly_pair_count	0	1	0	1	2	3	2	3
butterfly_input_count	0	0	0	0	0	0	0	0
stage_count	0	0	0	0	0	0	0	0

Number radix-4 butterfly calc	2				3			
Number radix-2 butterfly calc	8	9	10	11	12	13	14	15
butterfly_pair_count	0	1	0	1	0	1	0	1
butterfly_input_count	0	0	0	0	1	1	1	1
stage_count	2	2	2	2	2	2	2	2

Table D.7: Illustration of the pattern of *input_pair_ctrl* when the *fft* is calculation a 16 point FFT.

The *twiddle_addr_calc*s address control signal *twiddle_stage* is also created in *input_pair_ctrl*. The value of *twiddle_stage* is updated every time *stg_strobe* is set to 1 as can be seen in Table D.8.

stg_strobe	0	1	0	1
stg_ctrl	0	0	1	1
twiddle_stage	0	+1	0	-1

Table D.8: Illustration of the pattern of *twiddle_stage*.

Additionally when *butterfly_pair_count* is equal to *butterfly_pair_max* the *twiddle_stage* is incremented with 2.

The signal *fft_done* is the control signal that terminates *fft* and it is generated when the signal *finish* equal to 1 which it is when the FFT is finished with it's final stage.

The *input_pair_ctrl* was tested much like the *data_addr_calc* module was. By creating a testbed which emulates the input signals to the *input_pair_ctrl* module from the *fft* module. The output signals *butterfly_input_count*, *butterfly_pair_count*, *twiddle_stage* and *fft_done* are then cross-referenced with the Tables D.7 and D.8. The test found that the signals worked as intended.

D.2.4.3 Twiddle value access control

The twiddle factor function is split up in two separate modules, one for calculation the address of the twiddle factor and one that takes this address and loads the corresponding twiddle factor values to the output. However, since they are co-dependant it has been chosen that they should be described together.

$$w(n) = \exp\left(-j2\pi^{n}/_{N_{\mathrm{FFT}}}\right) \tag{D.9}$$

Where:

- $w(n)$ is a complex valued twiddle factor output at index $n \in (0, 1, 2, ..., {}^{N_{\mathrm{FFT}}}/_{2}$
- N_{FFT} are the total FFT length

The twiddle factor is defined in Equation D.9 but since the FPGA does not support exponential or complex functions easily, steps has to be taken to obtain the desired result output. Two main solutions has been considered:

- Since Equation D.9 describes points on the unit circle it is possible to calculate them by using a Taylor series expansion of the sine function.
- Another way of obtaining the twiddle factors is to store them in a lookup table.

The Taylor polynomial for a sine is:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \frac{x^{11}}{11!}... \tag{D.10}$$

For which different degrees are illustrated in Figure D.10.

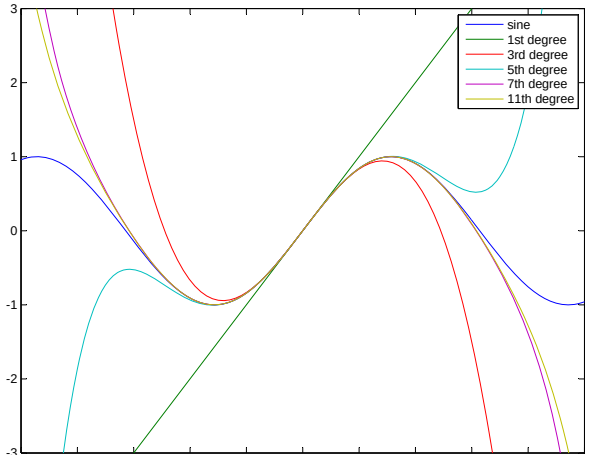


Figure D.10: Illustration of the Taylor series of a sine function described in Equation D.10. As seen on the figure the Taylor series are most effective within the first curve, whereafter many degrees are needed before any noticeable effect. Furthermore it is clear that a series of 5th order is needed to obtain a full curve.

As seen on Figure D.10 a polynomial of 5th degree is needed to obtain a full period of the sine function, which is needed to be able to calculate the twiddle factor. This means that it will be somewhat resource demanding, with respect to multipliers, if the twiddle factor were to be implemented as a calculation.

If a lookup table is used, the amount of required memory can be greatly reduced by taking advantage of the fact that the real and imaginary outputs on the unit circle are mirror versions of each other. Furthermore, the positive and negative value are also mirror versions of each others. This means that the lookup table only needs to consist of values in the range:

$$w = \sin\left(\frac{-\pi k}{N_{FFT}/2}\right) \qquad k \in 0, 1, ..., \frac{N_{FFT}}{4} \tag{D.11}$$

Which for an 2048 point FFT is equal to 512 values of 15 bit, since the sign value can be added based on the address.

The way to use the values in the registers will be to first check if the desired twiddle factor has a positive or negative real part. Secondly it must be check whether the the real and imaginary numbers are increasing or decreasing in value.

It has been chosen that the twiddle factor should be stored in memory registers, since it is important to reduce the number of multipliers used. The

```
7   reg [14:0] rom [0:511];
8
9   initial begin
10    $readmemb("C:/P9/P9_projekt/usrp2_utils/verilog/fft_include/twiddle_fact.
          txt",rom);
11  end
```

Listing D.5: Code of the *twiddle_rom* which initializes the memory registers and loads the data into the memory.

```
13  always @ (twiddle_addr) begin
14    case (twiddle_addr[9])
15      0: begin
16           w_r = {1'b0, rom[twiddle_addr[8:0]]};
17           w_i = {1'b1, (~rom[~twiddle_addr[8:0]] + 1)};
18         end
19      1: begin
20           w_r = {1'b1, (~rom[~twiddle_addr[8:0]] + 1)};
21           w_i = {1'b1, (~rom[twiddle_addr[8:0]] + 1)};
22         end
23    endcase
24  end
```

Listing D.6: Code of the *twiddle_rom* which uses the data in the memory to output the real and complex value of the twiddle factor.

module which holds the values are called *twiddle_rom*, and has two key functions; loading the data in the FPGA memory, and outputting the complex values of the twiddle factor correctly. The memory loading is done by the code seen in Listing D.5.

Note that when the data is loaded into the register, it is only 15 bit wide, this is due to the fact that the sign bit is created in the later step.

The output of the module is created by a case which first checks whether the input is on the positive or negative side of the real axis and afterwards outputs the two values mirrored, as seen in Listing D.6.

As seen in Listing D.6 the check is done by testing on the MSB, since this will be 1 if the real value is on the negative side and 0 if the real value is positive. The data access is done by the remaining bits, which is used to access the FPGA ROM. As seen this system uses dual access memory to spare FPGA fabric. It is easy to make this system use single access ram, however it will double the memory cost due to the fact that two lookup tables are needed.

To access the twiddle factor correctly, a module has been created which calculates the twiddle factor needed at a given address. The module is named *twitter_addr_calc* and its I/Os are seen in Figure D.11.

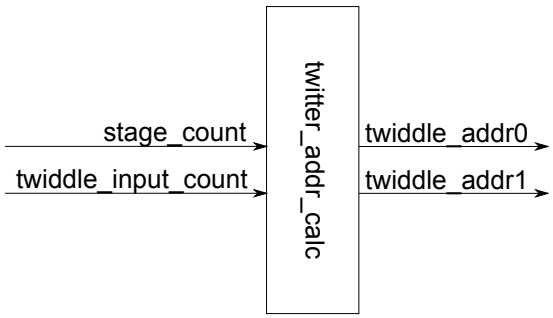


Figure D.11: In- and outputs of the *twitter_addr_calc* module, which handles the twiddle register access addresses of the *fft* module.

The module itself uses a single multiplier to calculate the twiddle address, based on the values of the inputs; *stage_count* and *twiddle_input_count*.

Test To test this system, a testbed has been made which loads the two modules and has a generator which mimics the actual system. The testbed is named *twiddle_tb.v* and the output of the *twiddle_rom* is compared to the output of the MATLAB script *twiddle_tb.m*.

To mimic the signals for the test, the signals for a 8 point FFT has been taken as basis, which means that the inputs will follow the flow seen in Table D.9

step	*stage_count*	*twiddle_input_count*	MATLAB output
0	0	0	0
1	0	0	0
:	:	:	:
7	0	0	0
8	1	0	0
9	1	1	-j
10	1	0	0
11	1	1	-j
:	:	:	:
15	1	1	-j
16	2	0	0
17	2	1	$\sqrt{2} - \mathrm{j}\sqrt{2}$
18	2	2	-j
19	2	3	$-\sqrt{2} - \mathrm{j}\sqrt{2}$
20	2	0	0
:	:	:	:
23	2	3	$-\sqrt{2} - \mathrm{j}\sqrt{2}$

Table D.9: Illustration of the test environment for the test of the two modules which controls which twiddle factor is to be used.

```
30  ////////////////////////////////////////////////////////////////////
31  //  Butterfly stage                                               //
32  ////////////////////////////////////////////////////////////////////
33  always @ (posedge do_stb) begin
34    y_temp_0 <= -yr_in * wr + yi_in * wi;
35    y_temp_1 <= -yr_in * wi - yi_in * wr;
36  end
37
38    assign xr = {xr_in[15], xr_in} - {y_temp_0[30], y_temp_0[30:15]};
39    assign xi = {xi_in[15], xi_in} - {y_temp_1[30], y_temp_1[30:15]};
40    assign yr = {xr_in[15], xr_in} + {y_temp_0[30], y_temp_0[30:15]};
41    assign yi = {xi_in[15], xi_in} + {y_temp_1[30], y_temp_1[30:15]};
42
43 endmodule
```

Listing D.7: The module *butfly_calc* which conducts a full radix-2 butterfly calculation. As seen in the code the intermediate result is ensured to be calculated first by the use of the fork-join statement.

During the test it was seen that the twiddle factors were output as desired, which leads to the conclusion that the modules works as intended.

D.2.4.4 Single radix-2 butterfly calculation

The butterfly calculation module is responsible for calculating the butterfly operation described in Equation D.3 to D.8 in Section D.

Since the calculation requires an intermediate result, it will happen in two sequential steps, which is also illustrated in Figure D.12 which describes the I/O relations.

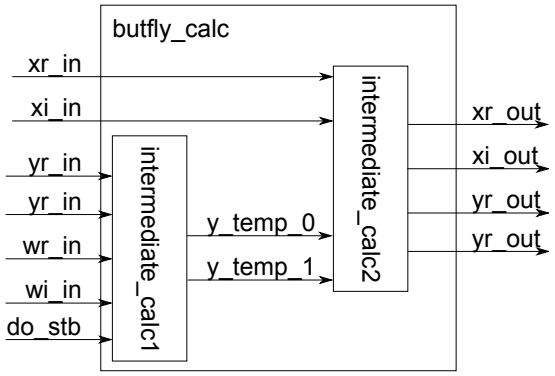


Figure D.12: In- and outputs of the *butfly_calc* module. Note that the two blocks called "intermediate calc" are not modules, just an indication of the fact that the two operations happens sequentially.

As seen in Listing D.8 the code of this module follows Equation D.3 to D.8 from Section D closely.

There have been two key points of focus in this module, whereas the first is to ensure that the intermediate result gets calculated before the output, the design takes advantage of the fork-join statement. Everything written between the fork and join keyword, will be executed in parallel, whereas the two fork-

join statements are executed sequential.

The second point to notice when designing this calculation module is how the data types end up after calculation. The input data type is defined by the *dsp_core* module of the USRP2 image to be one 32 bit register, containing two 16 bit 2's complement numbers describing the real and imaginary part of the sample. As seen in Listing D.8 the input to the FFT is being split up before entering the *butfly_calc* module.

The first part to analyse is the temp value calculation, which consist of two multiplications and one addition. Since the data type is 2's complement and the *dsp_core*'s out- and input utilises the full range of the 16 bit, it is important to check if there is a risk of overflowing when doing the calculations. Firstly, the *y_temp_0* and *y_temp_1* will be checked for worst case scenario outputs:

The input value of the variables *yr_in* and *yi_in* can range from $\pm 1 \pm \mathrm{j}$ and the twiddle factor can be in the range of $\pm\sqrt{2} \pm \mathrm{j}\sqrt{2}$, due to the fact that it lies on the unit circle. To check the maximum output values, the different extremities are checked and listed in Table D.10.

yr_in	*yi_in*	*wr_in*	*wi_in*	*y_temp_0*	*y_temp_1*
1	0	1	0	-1	0
1	0	$\sqrt{2}$	$-\sqrt{2}$	-1.41	1.41
1	0	0	-1	0	1
1	0	$-\sqrt{2}$	$-\sqrt{2}$	-1.41	1.41
1	1	1	0	-1	-1
1	1	$\sqrt{2}$	$-\sqrt{2}$	-2.82	0
1	1	0	-1	-1	1
1	1	$-\sqrt{2}$	$-\sqrt{2}$	-2.82	0
0	1	1	0	0	-1
0	1	$\sqrt{2}$	$-\sqrt{2}$	-1.41	-1.41
0	1	0	-1	-1	0
0	1	$-\sqrt{2}$	$-\sqrt{2}$	-1.41	-1.41
-1	1	1	0	1	-1
-1	1	$\sqrt{2}$	$-\sqrt{2}$	0	2.82
-1	1	0	-1	-1	-1
-1	1	$-\sqrt{2}$	$-\sqrt{2}$	0	-2.82
-1	0	1	0	1	0
-1	0	$\sqrt{2}$	$-\sqrt{2}$	1.41	-1.41
-1	0	0	-1	0	-1
-1	0	$-\sqrt{2}$	$-\sqrt{2}$	1.41	-1.41
-1	-1	1	0	1	1
-1	-1	$\sqrt{2}$	$-\sqrt{2}$	2.82	0
-1	-1	0	-1	1	-1
-1	-1	$-\sqrt{2}$	$-\sqrt{2}$	2.82	0
0	-1	1	0	0	1
0	-1	$\sqrt{2}$	$-\sqrt{2}$	1.41	1.41
0	-1	0	-1	1	0
0	-1	$-\sqrt{2}$	$-\sqrt{2}$	1.41	-1.41
1	-1	1	0	-1	1

Continued on next page

Table D.10 – continued from previous page

yr_in	*yi_in*	*wr_in*	*wi_in*	*y_temp_0*	*y_temp_1*
1	-1	$\sqrt{2}$	$-\sqrt{2}$	0	2.82
1	-1	0	-1	1	1
1	-1	$-\sqrt{2}$	$-\sqrt{2}$	0	2.82

Table D.10: Analysis of the outer extremeties of the output value of *y_temp* in the *butfly_calc* module of the *fft* module implementation.

As seen in Table D.10, the maximum absolute value which the *y_temp_0* and *y_temp_1* can be are 2.82, resulting in that the result will have a risk of overflowing with 2 bits given the current data representation.

The output of the *butfly_calc* will, due to the fact that *xr_in* and *xi_in* can take the value from $\pm 1 \pm \mathrm{j}$ and is added to the *y_temp_0* and *y_temp_1*, be able to take the maximum absolute value of 3.82. A value of 3.82 will still only lead to a 2 bit overflow, hence this is to take account of when scaling the outputs.

The scaling of the inputs has been chosen to be done by dividing by two, instead of four. This will result in the risk of an overflow, but it will be minimal since the inputs to the IFFT on the transmitter side will always be 1. Furthermore, using the scale factor of 2 for each butterfly output is consistent with the formular for making an IFFT using the FFT and imaginary-real switch.

The onboard multipliers in the FPGA does support multiplication of two values with precision up to 18 bit 2's compliment, which means that this is more than enough for the desired multiplication of two values of length 16 bits. When the synthesizer notices that two signed registers of length less than the 18 bits, it automatically handles sign extension and the output will have one sign bit and the length:

$$N_{mult\ out} = N_{mult1\ in} + N_{mult2\ in} - 1 \tag{D.12}$$

As seen in the code in Listing D.8 the division by two is handle by a sign extension of both the *x_in* and *y_temp* variable. Furthermore it is clear that the result of *y_temp* variable is full length, whereafter the MSB are chosen as output. The number of bits outputted by the multiplier is only 31, due to the fact that the Verilog compiler always takes the MSB's and put it into the result, no matter how long it is.

D.2.5 Control Logic

Having all of the modules designed and tested to fulfil their requirements, it is possible to use them in the FFT program. This is done by connecting them using the wires and registers seen in Table D.5 and as seen on Figure 5.6.

This section contains the code which controls the signal data flow using the clock master clock to set the different control signals. The explanation of the flow can be seen in the code.

```
262   always @ (posedge clk) begin  // 1 p
263     if (do_fft) begin           // if the FFT is not finished
264         input_strobe <= 0;      // ensure that input strobe is 0
265         dp_strobe0_o <= 1;      // call for new data 0 (x_in)
266         dp_strobe1_o <= 1;      // call for new data 1 (y_in)
267         stg_strobe <= 0;        // reset stg_strobe
268       @ (negedge clk)       // 1 n
269         get_new_but0 <= 1;      // calculate the first Butterfly
270         dp_strobe0_o <= 0;      // reset dp_strobe
271         dp_strobe1_o <= 0;      // reset dp_strobe
272         input_strobe <= 1;      // choose next addresses by going to the next
              state in the input_output_ctrl module
273       @ (posedge clk)       // 2 p
274         dp_strobe0_o <= 1;      // call for new data 0 (x_in) at the new
              addresses
275         dp_strobe1_o <= 1;      // call for new data 1 (y_in) at the new
              addresses
276         x_s1_b0 <= X0;          // store intermediate values of the first
              butterfly (32 bit number [real, imag])
277         y_s1_b0 <= Y0;          // store intermediate values of the before
              calculated butterfly
278         get_new_but0 <= 0;      // reset get_new_but signal
279         input_strobe <= 0;      // reset input_strobe
280       @ (negedge clk)       // 2 n
281         get_new_but1 <= 1;      // calculate next Butterfly
282         dp_strobe0_o <= 0;      // reset dp_strobe
283         dp_strobe1_o <= 0;      // reset dp_strobe
284         stg_strobe <= 1;        // ask for new twiddle values of the
              input_output_ctrl module
285         input_strobe <= 1;      // ask for new addresses (now ready to store
              first output)
286       @ (posedge clk)       // 3 p
287         x_s1_b1 <= X1;          // store intermediate values of the second
              butterfly (32 bit number [real, imag])
288         y_s1_b1 <= Y1;          // store intermediate values of the second
              butterfly (32 bit number [real, imag])
289         get_new_but1 <= 0;      // prepare for new calculation
290         ctrl_int1_ext0 <= 1;  // fft will now use the internal registers X0,
              X1, Y0, Y1
291         stg_strobe <= 0;        // reset stg_strobe
292         input_strobe <= 0;      // reset input strobe
293       @ (negedge clk)       // 3 n
294         get_new_but0 <= 1;      // calculate using the first radix-2 algorithm
              (parallel with the above)
295         get_new_but1 <= 1;      // calculate using the second radix-2 algorithm
               (parallel with the above)
296         we0 <= 1;               // set the write enable for the ram
297         we1 <= 1;               // set the write enable for the ram
298       @ (posedge clk)       // 4 p
299         dp_strobe0_o <= 1;      // store data at the first addresses (direct
              output of butfly_calc module)
300         dp_strobe1_o <= 1;      // store data in the first addresses (direct
              output of butfly_calc module)
301         x_s2_b0 <= X0;        // for debugging
302         y_s2_b0 <= X1;        // for debugging
303         x_s2_b1 <= Y0;          // store output of second output
304         y_s2_b1 <= Y1;          // store output of second output
305         get_new_but0 <= 0;      // reset get_new_but0
306         get_new_but1 <= 0;      // reset get_new_but1
307         ctrl_int1_ext0 <= 0;  // chose external output
308         /* fft done, only storing now. */
309       @ (negedge clk)       // 4 n
310         dp_strobe0_o <= 0;      // reset dp_strobe
311         dp_strobe1_o <= 0;      // reset dp_strobe
312         input_strobe <= 1;      // new addresses - store the second data pair.
313         mem1_direct0 <= 1;      // choose that the output to External ran
              should now be x_s2_b1 and y_s2_b1
314         stg_strobe <= 1;        // ready twiddle factors for next 4-point fft
315       @ (posedge clk)       // 5 p
316         input_strobe <= 0;      // reset input_strobe
317         dp_strobe0_o <= 1;      // store data at new addresses
318         dp_strobe1_o <= 1;      // store data at new addresses
319       @ (negedge clk)       // 5 n
320         input_strobe <= 1;      // get new addresses
321         we0 <= 0;               // no more write
322         we1 <= 0;               // no more write
323         dp_strobe0_o <= 0;      // reset dp_strobe
324         dp_strobe1_o <= 0;      // reset dp_strobe
325         mem1_direct0 <= 0;      // chose to det direct inputs
326     end else
327   end
```

Listing D.8: Control signal flow of the final *fft* module, as is written in verilog.