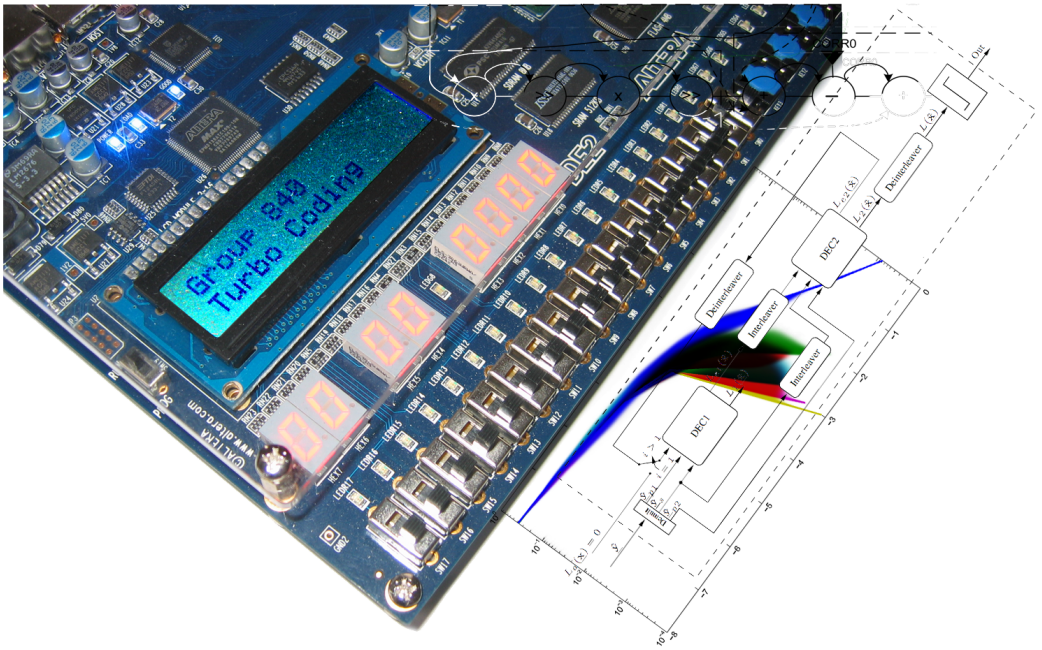
Evaluation of FPGA based Turbo Coding Implementations

- on a soft-core processor with hardware acceleration



AALBORG UNIVERSITY

Applied Signal Processing and Implementation (ASPI)
Group 840, 8th semester, Spring 2009
Faculties of Engineering, Science and Medicine
Department of Electronic Systems, Aalborg University

AALBORG UNIVERSITY

Applied Signal Processing and Implementation (ASPI)
Department of Electronic Systems
8th semester
Fredrik Bajers Vej 7
Phone 96 35 86 90
http://es.aau.dk

Title:
Evaluation of FPGA based Turbo Coding Implementations
- on a soft-core processor with hardware acceleration

Theme:
DSP Algorithms and Architectures

Project period:
ASPI8, Spring 2009

Project group:
09gr840

Group members:
Andreas Corneliussen
Erik Bundgaard Poulsen
Pradeep Silpakar
Troels Torkil Østeraa

Supervisor:
Yannick Le Moullec

Publications: 6

Pages: 90 (119 including appendices)

Finished: 2nd of June, 2009

Abstract:

This report serves as documentation for a project evaluating an FPGA based implementation of Turbo coding using both a soft-core processor and a soft-core processor aided by a hardware accelerator.

Turbo coding is a modern coding scheme that approaches Shannon's theoretical limit for the maximum information rate possible on a given channel and is used in many applications. The Turbo principle is based on iterative decoding with multiple soft-input/soft-output decoders.

In this project, the selected encoder is a parallel concatenation of two RSC encoders, while the decoder is based on two identical SOVA decoders.

A prototype is constructed using MatLAB to simulate the concept of Turbo coding, and to ensure that the Turbo coding functionality exists in the set of selected algorithms. Simulations of the prototype show that the BER decreases for every iteration in the decoder, except in cases with low SNR.

The aim of the project is to compare two implementations of a Turbo decoder. The first implementation is pure software executing on a Nios II/f soft-core processor, while the second implementation includes hardware acceleration.

Computationally demanding parts of the decoder are rewritten in VHDL, and thereby a large number of the most computationally intensive operations are moved from the software implementation to the hardware accelerator.

Decoding in the hardware accelerated implementation results in approximately the same BER as for the software implementation, but the execution time is decreased by between 34 % and 25 %, when the number of decoding iterations are increased from 1 to 20.

Preface

This project report is written by group 840 at the Department of Electronic Systems at Aalborg University as documentation for a project completed in the spring of 2009.

The report is mainly written for our supervisor and examiner, but also for other persons who may be interested in the subject.

Reading directions

This report is divided into three parts: Introduction and Analysis, Design and Implementation, and Conclusion and Discussion. The first part contains the introduction and an analysis of Turbo coding. Part two deals with the design of the system and performing the various implementations, and finally testing and comparing these. Part three contains a summarising conclusion, whereafter the achievements of the project are discussed.

In the introduction, a model that should guide the reader through the report, is presented. The model will be updated frequently in the subsequent chapters, and the pending step will be highligted in blue.

Citations

In this report, citations to the works of others are done using the author's last name and the year of publication presented in square brackets and separated by a comma. A citation can therefore look like this: "Shannon's theorem was firstly formulated in [Shannon, 1948]". A full bibliography list can be found on page 123.

Abbreviations

Abbreviations will be used throughout this report. The first time an abbreviation is used in each chapter it will be defined like, for instance: "Turbo Coding (TC) ". A list of all abbreviations used in this report can be found on page 124.

Figures and tables

A list of all figures and tables in this report can be found on page 127 and 129, respectively.

CD

A CD is attached to the last page of the report. Developed software, simulation results and other relevant material can be found on this CD.

This project report is written by:

Andreas Corneliussen	Erik Bundgaard Poulsen
Pradeep Silpakar	Troels Torkil Østeraa

Contents

Preface iii

Contents v

I Introduction and Analysis 1

1 Introduction 3
1.1 A^3 framework . . . 4
1.2 Modified A^3 framework . . . 5
1.3 Basic communication and coding theory . . . 6

2 Turbo encoder 13
2.1 Encoder structures . . . 14
2.2 Intermediate conclusion . . . 16

3 Interleaver 19
3.1 Purposes of interleaving . . . 20
3.2 Interleaving methods . . . 21
3.3 Comparative simulations . . . 26
3.4 Intermediate conclusion . . . 30

4 Turbo decoder 33
4.1 Assumptions . . . 33
4.2 Soft-input and soft-output . . . 33
4.3 Iterative decoding . . . 34
4.4 Choosing a decoding algorithm . . . 35
4.5 Soft Output Viterbi Algorithm . . . 36
4.6 Intermediate conclusion . . . 44

5 Prototype and compliance test 45
5.1 Introduction to the prototype . . . 46
5.2 Results . . . 47
5.3 Intermediate conclusion . . . 48

II Design and Implementation 49

6 Software implementation 51
6.1 C program specifics . . . 52
6.2 Platform specifics . . . 57

7 Applying Hardware Acceleration **63**
7.1 Profiling the soft-core implementation 64
7.2 Identifying part suitable for acceleration 66
7.3 Constructing the hardware accelerator 76

8 Comparison **81**
8.1 Test specification 81
8.2 Results 82
8.3 Intermediate conclusion 84

III Conclusion and Discussion **85**

9 Conclusion **87**

10 Discussion **89**

IV Appendix **91**

A Derivation of the extrinsic LLR **93**

B Software for Nios II soft-core processor **95**
B.1 `turbo_coding.h` 95
B.2 `main.c` 96
B.3 `decoder.c` 101
B.4 `sova.c` 102
B.5 `hw_sova.c` 106
B.6 `interleave.c` 110
B.7 `deinterleave.c` 111
B.8 `math.c` 111

C VHDL source code for hardware accelerator **113**
C.1 `bcc.vhd` 113

Bibliography **121**

List of Abbreviations **125**

List of Figures **125**

List of Tables **129**

Part I

Introduction and Analysis

1 Introduction

In 1948 C. E. Shannon determined a theoretical limit for the amount of information possible to transmit on a given channel per time unit[Shannon, 1948]. He proved this statement, not by use of a specific code, but by providing a mathematical proof that such a code must exist. Turbo Coding (TC) is a coding scheme in which it is possible to achieve results close to the limit proven by Shannon.

The concept of TC is the idea of using two or more codes on the same symbol sequence, and then use the knowledge obtained by decoding one code to improve the decoding of the second code and vice versa in an iterative manner. This means that, given two very different codes wherein all the information from the source data is preserved, TC is potentially able to iteratively decode until no more errors exist in the decoded symbol sequence. The error correcting performance of TC is therefore dependent on the difference between the two codes. One, frequently used method to assure high difference between the two codes is to interleave the input sequence to one of the encoders so that it encodes the same bits, but in a different order. A typical TC scheme therefore consists of three elements; encoder, decoder and interleaver (and de-interleaver).

The uses for TC are many because it approaches the Shannon limit. It is, for instance, used in applications like magnetic/optical data storage systems, ADSL modems and satellite communication [Sripimanwat, 2005].

However, because of the iterative behaviour of the decoding process, the error correcting performance is bounded by the computational power and precision of the platform on which it is implemented [Jin et al., 2006]. It can therefore be hard to implement TC on embedded platforms and use the concept to the full potential, because such platforms are often limited in computational power and precision.

The objective of this project is to test if it is possible to aid the implementation of TC on an embedded platform (a full software solution) by use of hardware acceleration. This is done by placing a soft-core processor on an FPGA and then designing an implementation of TC for this processor. Afterwards it is attempted to decrease the execution time of this implementation by use of hardware acceleration.

The question to be answered in this project can be formally written as the following project statement:

> *"Is a combined soft-core and hardware implementation better or worse than a pure soft-core implementation with regards to execution time at different numbers of decoding iterations?"*

In order to describe the process in the project, it is beneficial to introduce a framework that models the process of designing a digital signal processing algorithm for a given application and implementing this on a platform.

1.1 A^3 framework

The framework used in this project is called the A^3 framework model, and has been developed at Aalborg University.

The framework consists of the three domains depicted in Figure 1.1 by circles; Application, Algorithm and Architecture. One needs to consider each domain and the mappings between them, to obtain a well-designed product.

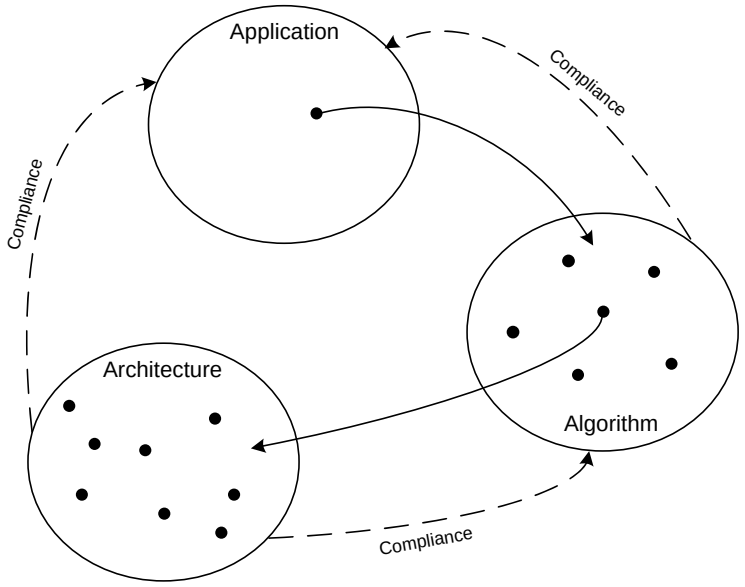


Figure 1.1: The A^3 framework model. Different algorithms are found to fulfil a single application, where after the algorithm of choice may be implemented on multiple types of architectures. The dashed arrows ensure compliance between the different modules so that the implementation in the end will fulfil the specification of the application.

For each application in the Application domain there are multiple possible algorithms that fulfil the specification of this application. Likewise are be multiple possible implementations for each algorithm. Therefore all mappings between domains are 1:many mappings. For instance, algorithms can be mapped onto multiple kinds of architectures, for example Harvard, Von Neumann or a custom architecture where for example the number of Multiply & Accumulate (MAC) operators or Arithmetic Logical Unit (ALU)s can be defined.

The framework is iterative, which basically means that tests must be completed to show compliance between the choices made in all domains and the project objective and one can use the knowledge obtained in, for instance, the architectural domain to improve the application specifications. Such objectives could for example be functional constraints, demands for timing or throughput or demands for low area or power consumption.

1.2 Modified A^3 framework

The framework just described is, however, developed to illustrate the process of a full system design. In this project the application is non-existing in the sense that it is decided that rather than identifying a problem where TC can be the solution, the project group has decided to implement TC in two different ways and compare them. It is therefore decided to modify the framework so that the first domain is changed from application to system functionality.

The modified framework, applied to the present state of the project, is illustrated in Figure 1.2. Here it can be seen that the functionalities that are to be implemented are known, as these are the three components that typically make up a TC scheme. The first task is therefore to analyse the mappings from these functionalities to the specific algorithms that are able to perform the tasks. This process is described in Chapter 2 to 4. Here each functionality is described and it is described which algorithms can accommodate these functionalities. In some cases a choice is made regarding which algorithm to choose in order to reduce the number of algorithms requiring analysis. Such choices might not be justified by better reasons than that it is the most common or simple algorithm, but as this report is written as a part of a learning situation, and because the focus of the project is on the relative execution time rather than algorithmic throughput, these are considered to be reasonable.

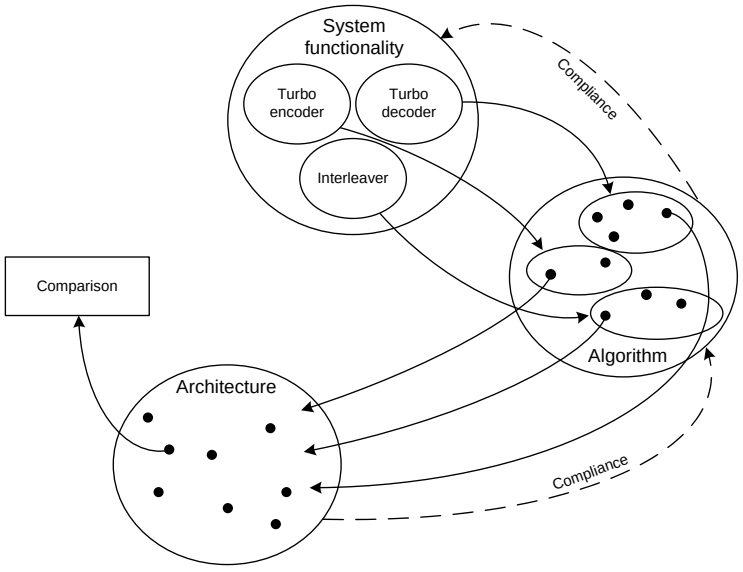


Figure 1.2: The modified A^3 framework model.

In Figure 1.2 it can also be seen that the compliance between a selected algorithm and the functionality it is to perform needs to be ensured by a compliance test. The algorithms described in Chapter 2 to 4 are therefore summarised into a prototype which is constructed in MatLAB,

and simulations are performed to ensure that the selected algorithms function as expected. This prototype and the simulations are discussed in Chapter 5.

The second part of the report is used for describing the mapping from the algorithmic domain to the architectural domain, while the last part of the report is dedicated to the evaluation and comparison of the two different implementations.

In order to understand the concept of turbo-coding, it is necessary to introduce some basic concepts of communication theory.

1.3 Basic communication and coding theory

A typical communication system is composed as depicted in Figure 1.3. A discrete information source, such as a microphone connected to an A/D-converter or a computer outputting discrete symbols directly, generates information meant to be communicated to some information sink on the other side of a communication channel. In order to remove redundancy in the stream of symbols output from the information source, a source coder is employed which is capable of removing redundancy in a way that is reversible, so that a decoder can recreate the redundant information. After the removal of all redundant information in the source coder, the channel coder will add some controlled redundant information again so that the channel decoder is able to correct or detect errors on the channel, and thereby improve the reliability of the system.

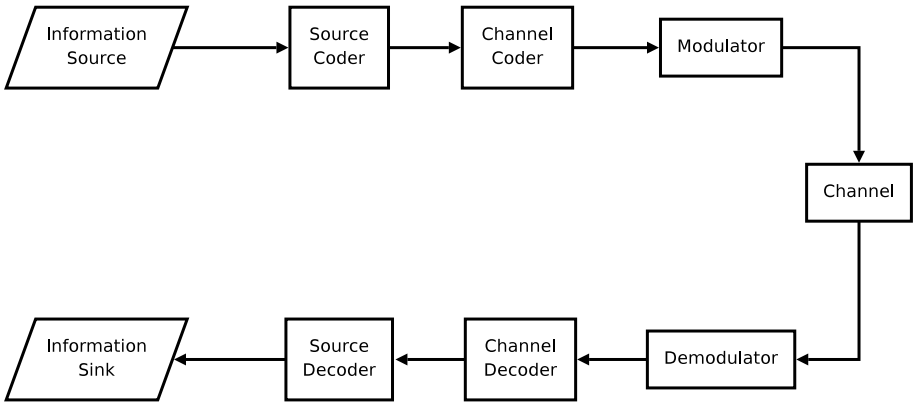


Figure 1.3: The components of a typical communication system [Proakis and Salehi, 2002, p. 8].

The modulator is in charge of changing the discrete symbols into signals more fit for transmission on the given channel. The effect of the channel on the modulated information is that the stream of signals can be altered in such a way that the demodulation process will introduce errors when trying to estimate which symbols have been emitted from the channel coder.

1.3.1 The AWGN channel model

A widely used model of the channel is that of Additive White Gaussian Noise (AWGN). In this model the signals transmitted on the channel are corrupted by addition of white Gaussian noise. This noise is described by the probability distribution given by [Haykin, 2001, p. 54]:

$$f_W(x) = \frac{1}{\sqrt{2\pi\sigma_W^2}} e^{\frac{-x^2}{2\sigma_W^2}} \tag{1.1}$$

where the variance σ_W^2 determines the power of the added noise. Figure 1.4 shows the distribution of such noise while Figure 1.5 shows an example realisation of white noise and Figure 1.6 shows the frequency spectrum of this realisation. Ideally AWGN has a totally flat frequency spectrum, but in order to achieve that, the sequence needs to have infinite length. Thus, AWGN exists only in theory, but it is a good approximation to many communication channels, and is therefore widely used to estimate the effect of noisy channels on an information system.

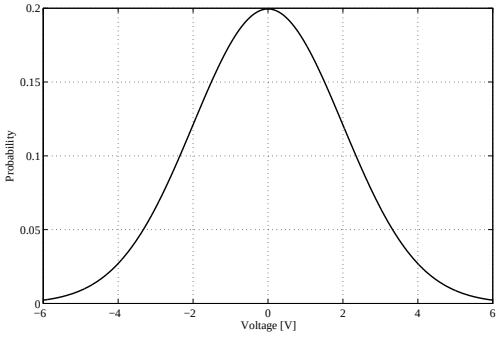


Figure 1.4: The Probability Density Function (pdf) (see 1.1) of Gaussian noise with variance 2.

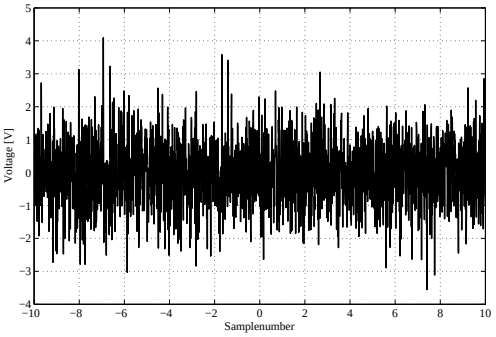


Figure 1.5: A realisation of white Gaussian noise with variance 2 (plotted by use of the Matlab-function `randn`).

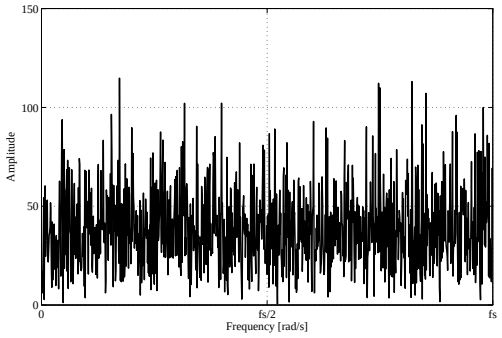


Figure 1.6: FFT of the realisation of white Gaussian noise depicted in Figure 1.5.

1.3.2 Simplified model

When, as it is the case in this project, only the effect of the channel coding is of interest, one can choose to raise the abstraction level so that the source coder and decoder become part of the information source and sink, respectively, while the modulator, demodulator and channel is combined into a discrete symbol channel.

When doing this, the channel is simplified into a set of probabilities that describe the probability of error. Furthermore the information source is now assumed to output each symbol with equal probability. The simplified model is depicted in Figure 1.7.

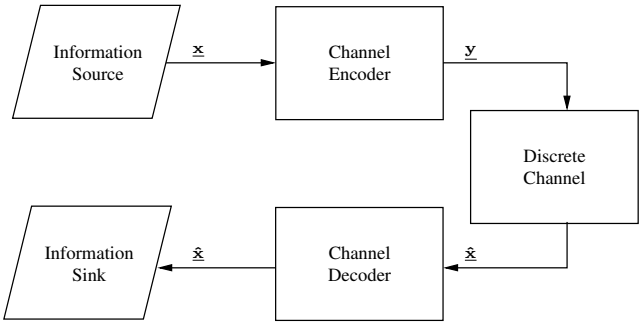


Figure 1.7: A simplified model of the communication system.

1.3.3 Channel Capacity

It is clear that if the probability of error in a channel is high, addition of redundancy can improve both the throughput and the reliability of the communication. The reliability is improved by using error detecting codes, and the throughput can be improved by reducing the number of retransmissions using error correcting codes.

By using Shannon's limit it is possible to calculate how much information can be transmitted over a specific channel. Shannon's limit is stated like this:

> Theorem 11: Let a discrete channel have the capacity C and a discrete source the entropy per second H. If $H \leq C$ there exists a coding system such that the output of the source can be transmitted over the channel with and arbitrarily small frequency of errors (or an arbitrarily small equivocation). If $H > C$ it is possible to encode the source so that the equivocation is less than $H - C + \epsilon$ where ϵ is arbitrarily small. There is no method of encoding which gives an equivocation less than $H - C$. [Shannon, 1948]

Where the capacity C of a channel is defined as:

$$C = \max(H(x) - H_y(x)) \tag{1.2}$$

Where:		
$\max$	is over the symbol probabilities.	
x	is the unknown input symbol	$[-]$
y	is the unknown output symbol	$[-]$
$H(x)$	is the average information from a source; $H(x) = -\sum_i^N p_i \log_2(p_i)$	$\left[\frac{\text{bit}}{\text{symbol}}\right]$
p_i	is the probability of the source outputting the i'th symbol at any given time	$[-]$
$H_y(x)$	is the equivocation of x given y. This is the knowledge of x still missing while knowing the value of y; $H_y(x) = H(x,y) - H(y)$	$\left[\frac{\text{bit}}{\text{symbol}}\right]$
$H(x,y)$	is the average combined information in knowing x and y; $H(x,y) = \sum_{i,j} p(i,j) \log_2 p(i,j)$	$\left[\frac{\text{bit}}{\text{symbol}}\right]$
$p(i,j)$	is the joint probability of x being the i'th symbol and y being the j'th symbol	$[-]$

This means that the channel capacity forms a boundary inside which it is possible to employ a coding scheme that results in virtually error-free communication. However, if the source outputs

more information than the channel can transfer, the amount of additional information cannot be transmitted without error.

1.3.4 Coding methods

The general purpose of Error Control Coding (ECC) in a communication system is, as mentioned, to attempt to achieve the error-free communication promised by Shannon's limit, if the source fits the channel. Generally, two types of ECCs exist - Forward Error Correction (FEC) and Automatic Repeat Request (ARQ) codes. The purpose of FEC codes is to correct errors introduced by the channel, and thereby avoid the need for re-transmissions. ARQ codes, on the other hand, are designed to detect errors, so that a requests for re-transmission can be sent. In many cases, both FEC and ARQ coding schemes are employed, FEC to raise the throughput and ARQ to make sure to detect errors that FEC could not correct [Wells, 1999, p.153]. The focus of this project is, however, solely on FEC.

Among others are two different types of FEC codes in common use today:

- Block codes
- Convolutional codes

These types are described next.

Block codes

Block codes are specified as (n,k) codes [Wicker and Kim, 2008, p. 2]. The encoder takes k information bits (called a message block) and computes $(n-k)$ parity bits using a code generator matrix. These parity bits are concatenated with the information bit to form a code word of n bits in total. This output code word depends only on the input message block, i.e, it is a memory-less encoder. Given a message length of n bits, there are 2^n possible sequences. However, if this sequence is the result of a (n,k) block code, only 2^k of these are valid codewords. If receiving an invalid codeword, the decoder can correct the sequence to the closest valid codeword, which in many cases is the correct one. If, however, too many errors have occurred, correcting using block codes can result in errors, as the many errors will make the received sequence look more like a wrong (but still valid) codeword.

For any of the 2^k possible message blocks, encoders will have an output range of 2^k different possible code words of length n bits. The ratio of the number of input bits to the number of output bits of encoder is known as code rate:

$$R = \frac{k}{n} \tag{1.3}$$

The parity bits provide the code with the capability of being decoded to obtain the original message, even though noise may have caused corrupted bits in the received sequence [Sweeney, 2002, p. 4].

Convolutional Codes

Convolutional coding has gained popularity in many communication applications because of the fact that linear block codes tend to need very large blocks in order to achieve low bit-error rates [Ratzer, 2001, p. 1]. An encoder for a convolutional code accepts an information sequence of k bits and produces an encoded sequence (codeword) of length n bits. The encoder operates on the incoming message sequence continuously in a serial manner. Unlike block coding, each encoded block depends not only on the corresponding k bits of input at the same time unit or index, but also on the m previous bits. This implies that the encoder has a memory of length m.

Convolutional codes are specified as (n,k,m), where n is the number of output bits from the encoder at a time, k is the number of input bits to the encoder at a time and m is the maximum number of shift register stages in the path to any output bit. Convolutional codes are also defined simply by the notation (R,K), where R is the code rate, expressed as in Equation (1.3), while the input constraint length $K = m + 1$ is the total number of bits involved in the encoding process.

The structure of a convolutional encoder of type (2,1,2) is illustrated in Figure 1.8. As noted, it features two memory registers and three modulo-2 adders to represent two output bits. The encoder illustrated in Figure 1.8 has the rate $R = 1/2$ and the constraint length $K = 3$.

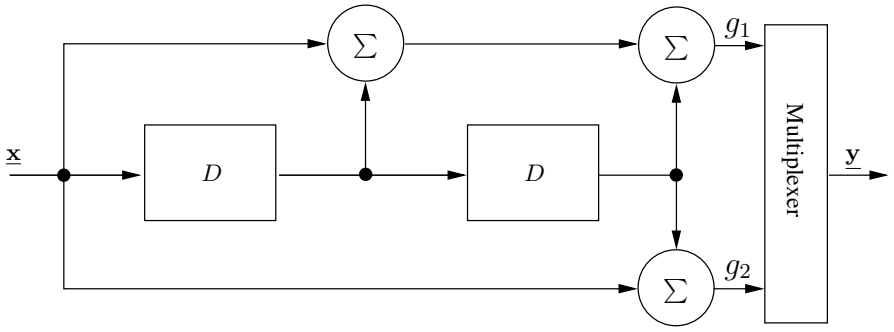


Figure 1.8: Convolutional encoder with $R = 1/2$ and $K = 3$

The selection of bits to be added, to produce the output code sequence, forms generator polynomials denoted by $g_n(D)$. A full set of the generator polynomials completely describe the convolutional encoder. Thus, the encoder shown in Figure 1.8 can be described with the two generators $g_1(D)$ and $g_2(D)$, as follows:

$$g_1(D) = D^2 + D + 1 \tag{1.4}$$

$$g_2(D) = D^2 + 1 \tag{1.5}$$

The output of this type of encoder is called Non-Systematic Convolutional (NSC) code as the input bit is not directly visible in the output stream. Depending on the configuration of the convolutional code, many choices of generator polynomials for an m order code can be constructed. However, not necessarily all generator polynomials will perform well with regards to error correction.[Fan Mo, 1999] .It is chosen on the by performing simulation.

The truth table for the convolutional encoder in Figure 1.8 is shown in Table 1.1.

Start State	Input	End State	Output
00	0	00	00
00	1	10	11
01	0	00	11
01	1	10	00
10	0	01	10
10	1	11	01
11	0	01	01
11	1	11	10

Table 1.1: Truth table for the NSC encoder illustrated in Figure 1.8. The states are numbered by the content of the two delay elements.

The structural properties of the convolutional code produced by the encoder in Figure 1.8 can be represented by a graphical description using a state diagram (illustrated on Figure 1.9) or a trellis diagram (illustrated on Figure 1.10).

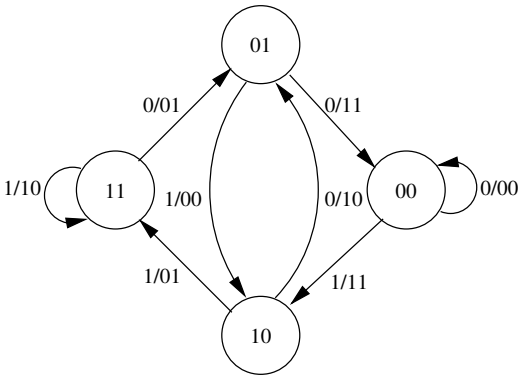


Figure 1.9: State diagram of the system depicted in Figure 1.8. The states are determined and named by the content of the two delay elements while the transitions are marked with input/output to the states.

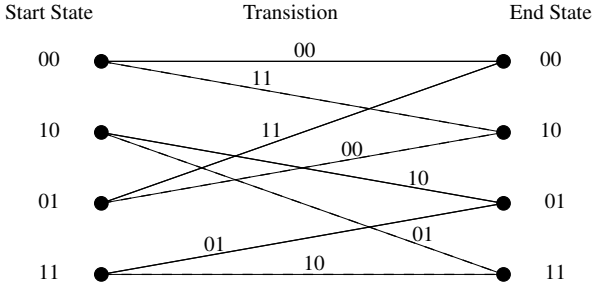


Figure 1.10: Trellis diagram of the system depicted in Figure 1.8. The transitions are represented by dashed lines for input sequence bits equalling 1 and solid lines for input sequence bits equalling 0. The transitions are marked with the corresponding output.

RSC Encoder

As mentioned, the encoder described above produces an NSC code. The Recursive Systematic Convolutional (RSC) encoder is obtained by providing a feedback from one or more of the shift registers into the input. Figure 1.11 illustrates this structure.

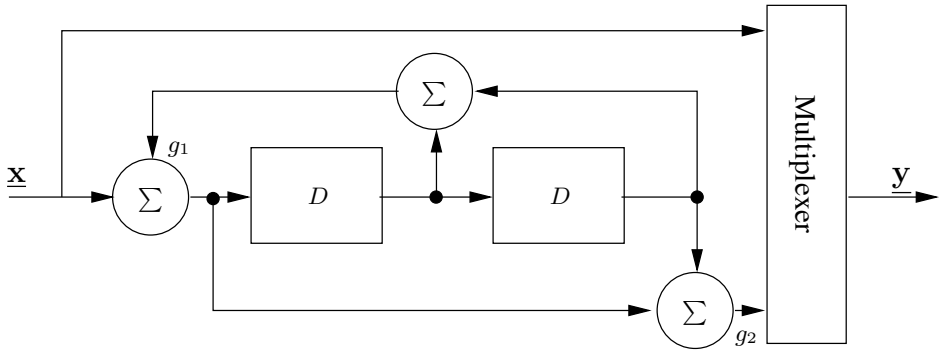


Figure 1.11: The RSC encoder with $R = 1/2$ and $K = 3$.

The generator of the code is designated by (1,g_2/g_1). For Figure 1.11 g_1 and g_2 has generator polynomial sequences of g_1=(111) and g_2=(101) respectively, which correspond to the numbers 7 and 5. The generator code can thus be written as (1,5/7). The 1 indicates that the data bit passes directly into the output code (hence the code is systematic), 5 indicates that the feed forward polynomial is $g_2(D) = D^2 + 1$ and 7 indicates the feedback polynomial $g_1(D) = D^2 + D + 1$. Hence, the first output g_1 is the feedback to the input of the encoder. The corresponding state diagram of the RSC encoder is shown in Figure 1.12.

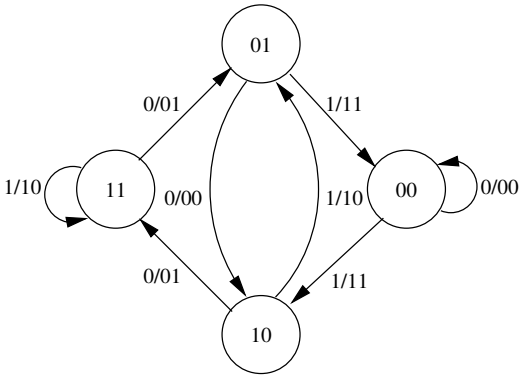


Figure 1.12: State diagram for the (1,5/7) RSC encoder depicted in Figure 1.11. The states and transitions are marked as on Figure 1.8.

Generally, the Bit Error Rate (BER) performance of RSC codes is better at low E_b/N_o (low Signal-to-Noise Ratio (SNR)) compared to that of NSC codes. However, at higher SNRs the roles are interchanged, i.e. the trend is that NSC achieves a lower BER than RSC [Sklar, 2002, p. 16].

With the basic theory behind channel coding and general theory of communication established, the purpose of the next three chapters is to discuss the basic building blocks of TC, and thereby describe the mapping from functionality to algorithm.

2

Turbo encoder

Having established the project objective, and the framework for reaching this objective, the first phase of the framework can begin. This phase is illustrated in Figure 2.1, and consists of the algorithmic perspective of turbo coding. In other words, in the following chapters, the basic building blocks of Turbo Coding (TC) are analysed, and it is discussed which algorithms are suitable. The first functionality to be analysed is the turbo encoder.

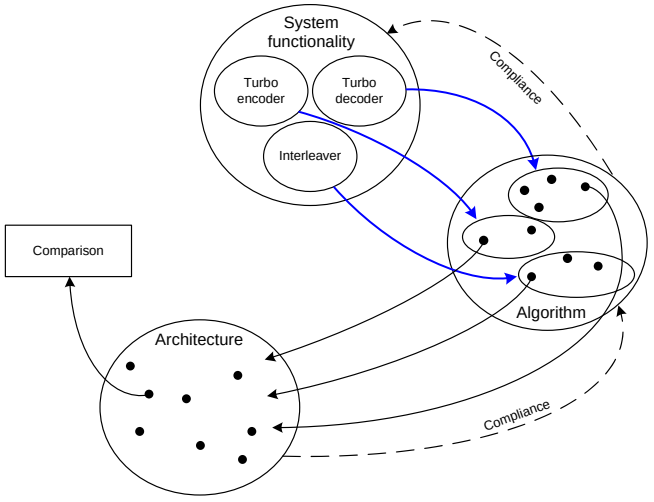


Figure 2.1: Illustration of the current phase of the project with regards to the modified A^3 model. This is the phase for analysis of the algorithms for the different blocks which together form a turbo coder.

2.1 Encoder structures

The encoder for TC is based on two or more convolutional encoders separated by an interleaver. In turbo coding, Recursive Systematic Convolutional (RSC) encoders are preferred over Non-Systematic Convolutional (NSC) encoders because the former gives better performance at low Signal-to-Noise Ratio (SNR)s. However, at higher SNRs, the opposite is generally the case, unless code puncturing and concatenation of multiple RSC encoders are used [Thitimajshima, 1995, p. 2267, 2269], [Sklar, 2002].

Regardless of the encoder type, the encoders can be organised in three possible ways:

- Parallel Concatenated Convolutional Code (PCCC)
- Serial Concatenated Convolutional Code (SCCC)
- Hybrid Turbo code

However, as the purpose of this project is to test if hardware acceleration is beneficial when implementing TC on an embedded platform, each of these are equally valid. It is therefore chosen to solely analyse the Parallel Concatenated Convolutional Code (PCCC) type.

It is necessary to delimit in this way as each of the encoder structures will require different decoder structures.

The parallel structure of PCCC uses two or more RSC encoders in parallel with an interleaver between each RSC encoder. To illustrate this, a basic turbo encoder using two RSC encoders in parallel with rate $R = 1/3$ (one input bit corresponds to three output bits) is shown in Figure 2.2. While there are no limit to the number of RSC encoders to concatenate, generally only two encoders are used because with the increase in the number of encoders, bandwidth efficiency is reduced and there is no proportionate decrease of the BERs at the decoder [Langton, 2006]. Thus, while Figure 2.2 may show an overall simple encoder, this type of encoder fulfills the turbo coding principle.

In designing the two encoders that are included in the turbo encoder it is necessary to ensure one of two things; Either the two codes output from the encoders must be very different so that when one of the encoders output a codeword of low quality, the other must output one of high quality, or the turbo encoder must include an interleaver as illustrated in Figure 2.2. Such an interleaver will make sure that the input-stream to the two encoders will differ, and the generated codes will therefore also differ. One can of course use both methods to ensure the difference of the outputs of the encoders.

The output of the encoder in this setup is formed by a combination of information bits, followed by the parity bits from the two RSC encoders.
These parity sequences are generated corresponding to a ratio of generator polynomials g_2/g_1, as already described in Section 1.3.4 on page 10. The first component encoder operates directly on the information bit sequence $\underline{\mathbf{x}}$ and produces two output sequences $\underline{\mathbf{y_{s1}}}$ and $\underline{\mathbf{y_{p1}}}$. The second component encoder operates on the $\underline{\mathbf{x}}'$ sequence bit, which is a permuted version of $\underline{\mathbf{x}}$, constructed by the interleaver. Similarly, it produces the output sequences $\underline{\mathbf{y_{s2}}}$ and $\underline{\mathbf{y_{p2}}}$.

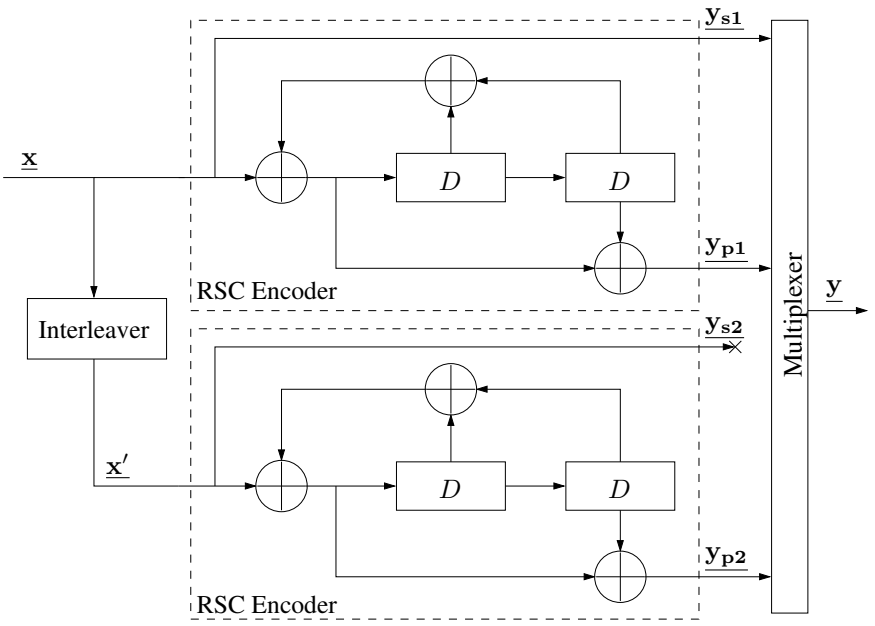


Figure 2.2: PCCC Turbo encoder with rate $R = 1/3$

For the encoder of Figure 2.2, which consists of two component encoders both having the same amount of memory $m_1 = m_2 = 2$, the following outputs are achieved:

$$\begin{aligned}
\underline{\mathbf{y_{s1}}} &= \underline{\mathbf{x}} \\
\underline{\mathbf{y_{p1}}} &= \frac{g_2}{g_1} \cdot \underline{\mathbf{x}} \\
\underline{\mathbf{y_{s2}}} &= \underline{\mathbf{x}'} \\
\underline{\mathbf{y_{p2}}} &= \frac{g_2}{g_1} \cdot \underline{\mathbf{x}'}
\end{aligned}$$

The systematic bit from only one encoder is transmitted. The reason for this is that the other systematic sequence is merely a permuted version of the one sent, and sending both will therefore be redundant. For each input, three bits are therefore sent if using the encoder illustrated at Figure 2.2, and this turbo encoder is therefore of rate 1/3.

Fundamentally, both RSC and NSC encoders can be used in a TC encoder. However, it is of some importance that the Hamming weights are distributed evenly on all input values. This means that an input sequence of low weight should correspond to an output with relatively high weight. As it can be seen on Figure 2.3 and Figure 2.4, the way a low weight input (in this case an impulse) is treated by the two types of encoders is quite different.

It is seen that the RSC encoder outputs a codeword of significantly higher weight, and RSC encoders will therefore be desirable when designing a turbo encoder for use in low SNR.

2.1.1 Trellis Termination

In order to ease the decoding of the data, it is beneficial to ensure that the encoder assumes a specific state at the end of each transmitted sequence. In this way the start and end state of

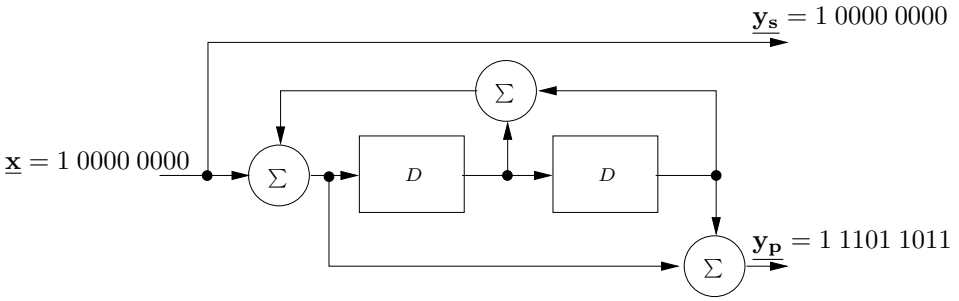


Figure 2.3: Recursive Convolutional encoder with an impulse as input. The systematic output is therefore also an impulse response.

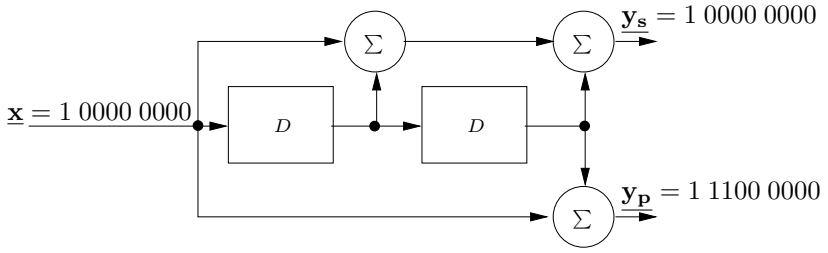


Figure 2.4: Non-recursive Convolutional encoder with an impulse as input. The systematic output is therefore also an impulse response.

the encoder is known by the decoder, so that it is easier to estimate which bit could have been transmitted.

The process of making the encoder reach the all-zero state (all registers in the structure are zero) is called trellis termination. In NSC encoders, trellis termination can be done by inputting the same amount of zeros as there are registers in the encoder structure. In the case of RSC encoders, trellis termination is done by copying the feedback value to the input and doing this for the same amount of bits as there are registers in the encoder structure.

2.1.2 Puncturing

When communicating over a channel with low SNR, a turbo encoder using component encoders with a low rate (more parity bits for each input bit) will perform well. However when communicating over a channel with high SNR, all the extra information introduced by the low rate is redundant and unnecessary.

It may not, however, be possible to change the component codes, which is why the concept of puncturing is introduced. Puncturing is done by removing some of the bits in the output-stream to increase the rate of the turbo encoder, and thereby reducing the unnecessary redundancy.

Puncturing is not used in this project, and the concept will therefore not be discussed further.

2.2 Intermediate conclusion

There are three possible encoding structures for turbo codes, and two types of component encoders can be used. The analysis in this chapter has been limited to the PCCC structure, while both the NSC and RSC encoder types were analysed and described. From the analysis of these encoder

types, the RSC encoder type is chosen as it distributes the Hamming weights better over all possible inputs. As stated in [Sklar, 2002], the code weights is an important measure of how well a turbo code performs with regards to the Bit Error Rate (BER) at different SNRs.

The concepts of trellis termination and code puncturing have been introduced, but code puncturing is not discussed further.

It was also determined that the interleaver plays an important role in increasing the difference between the two generated codes. The interleaver is therefore the subject of the analysis in the next chapter.

3
Interleaver

The purpose of this chapter is to analyse a few of the most common methods used for interleaving which is utilised in Parallel Concatenated Convolutional Code (PCCC) encoders and Turbo decoders. The function of the interleaver is also discussed, and thus why it should be implemented. From the analysis, a few simulations on the suitability of the different methods are carried out. From these, the most suitable method is found.

The process of interleaving basically means to permute a set of data from one sequence to another. In this chapter, the input to the interleaver is denoted with the vector $\underline{\mathbf{c}}$, and the output is denoted with the vector $\underline{\mathbf{c}}'$. The process of interleaving, or permuting, can be written as follows:

$$\{\underline{\mathbf{c}} = \left[\ c_i\ \right], \quad 0 \leq i < N\ \} \longrightarrow \{\underline{\mathbf{c}}' = \left[\ c'_j\ \right], \quad 0 \leq j < N\} \tag{3.1}$$

The indices i and j may not reference the same data element if $i = j$. For the point of illustration, the examples in this chapter will make use of decimal values as elements in the vectors, as examples using binary values are hard to follow. Furthermore, the input vector for all examples is the vector of increasing integers starting from zero.

An example illustrating the use of Equation 3.1 is:

$$\underline{\mathbf{c}} = \left[\ 0 \quad 1 \quad 2 \quad 3 \quad 4\ \right] \longrightarrow \underline{\mathbf{c}}' = \left[\ 3 \quad 1 \quad 4 \quad 0 \quad 2\ \right]$$

This mapping of indices can also be done by using a permutation matrix, or interleaver matrix, $\underline{\underline{\mathbf{P}}}$:

$$\underline{\mathbf{c}}'^{\mathrm{T}} = \underline{\underline{\mathbf{P}}} \cdot \underline{\mathbf{c}}^{\mathrm{T}} \tag{3.2}$$

The permutation matrix for the example above is:

$$\underline{\underline{\mathbf{P}}} = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix} \tag{3.3}$$

The process of interleaving is therefore described either by Equation 3.1 or by Equation 3.2. In many uses of an interleaver, and especially for the use in a Turbo Coding (TC) scheme, the interleaving process must be reversible. By noting that $\underline{\underline{\mathbf{P}}}$ is an orthogonal matrix, it can be seen that deinterleaving is done by:

$$\underline{\mathbf{c}}^{\mathrm{T}} = \underline{\underline{\mathbf{P}}}^{\mathrm{T}} \cdot \underline{\mathbf{c}'}^{\mathrm{T}} \tag{3.4}$$

The purposes of interleaving in the encoding processs is now discussed.

3.1 Purposes of interleaving

When transmitting data, it is normally influenced by noise, as described in Section 1.3.1. In the case where a code vector $\underline{\mathbf{c}}$ is transmitted, the amount of noise influences the amount of corrupted bits. Groups of sequential errors in a code must be avoided, because reconstruction of the sent signal in the decoder may then be impossible. In other words, a code with an evenly distributed number of erroneous bits perform better than a code containing the same amount of erroneous shifted bits, but consecutive [Sanchez et al., 2000]. This is best visualised by an example of errors in a short sentence. If the errors are distributed over the entire sentence, the person reading it, is most likely able to decode the message.

$$\underline{\mathbf{c}} = (\text{Thiq isTa sUortVtesw})$$

On the other hand, if the errors are grouped together the full message might be incomprehensible. The message in the sentence above is easier to understand than the message in the sentence below.

$$\underline{\mathbf{c}} = (\text{This is a shortQffmX})$$

This is due to the fact that the English language contains redundancy in such a way that the reader is able to read a message despite errors, but if the errors are grouped, entire words might be unintelligible, and the full message will therefore be uncertain.

The point of the interleaver is therefore to permute data in such a way that no (or only very few) bits retain their indices, and eventual burst errors are distributed on the whole code, and thereby improve the probability of the decoder to be able to correctly decode the message. In Recursive Systematic Convolutional (RSC) codes, the interleaving process has the property of reducing

the amount of code words with low Hamming weight, due to an effect called spectral thinning [Perez et al., 1996]. This improves the overall quality of the code.

Interleaving is furthermore used to ensure high difference between the output of the two component encoders. This makes the Turbo decoder more likely to be able to decode the message.

Three methods of interleaving are analysed and their suitability determined from an evaluation of the structure of the permuted code vector. In particular, the criteria proposed for permutation is summarised by the two constraints:

- The amount of index shifts between all bits should be random.
- The index shifts must be maximized for all bits in the code vector.

These two constraints define the ideal interleaver structure, and naturally a trade-off must be made to comply with both requirements.

3.2 Interleaving methods

Three different methods of interleaving a sequence of data are analysed and described, namely the golden interleaver, the S-random interleaver and the quadratic interleaver. After this, it is tested if all three methods comply with the two constraints just described, and it is considered which of these is most suitable in this project.

3.2.1 Golden interleaver

The golden interleaver is one of three methods all described in [S. Crozier, 1999]. All three methods rely on the property of the golden ratio, which explains their names. Only the golden interleaver of these three is analysed, since the golden prime interleaver is known to suffer from quantization problems and the golden dithered interleaver requires block sizes larger than about 1000 bits to function properly. The overall system designed in this project should be versatile with regards to the block length to be able to test the effect of different block lengths. Therefore the golden dithered interleaver is discarded.

As mentioned, the ”golden” term stems from the golden ratio, where a line of unity (1) is divided into two sections of length *a* and *1-a*. The ratio of *a* to the unit length must equal the length of the remainder to the length of *a* [Markowsky, 1992, p. 2]:

$$a = \frac{1-a}{a} = 0.6180 \tag{3.5}$$

In the golden interleaver, the permutation of the vector $\underline{\mathbf{c}}$ to $\underline{\mathbf{c}}'$ is determined by the following algorithm:

1. Select an order m of a.

2. Determine the value of

$$b = N \cdot \frac{(a^m + \psi)}{r} \tag{3.6}$$

Where,
N is the length of the code vector $\underline{\mathbf{c}}$
ψ is any integer modulo r
r is an integer influencing how far apart the $\underline{\mathbf{c}}(i)$ and $\underline{\mathbf{c}}'(j)$ will be in indices.

3. Determine the index mapping vector $\underline{\mathbf{v}(\mathbf{n})}$ as:

$$\underline{\mathbf{v}(\mathbf{n})} = \epsilon + n \cdot b \bmod N \tag{3.7}$$

Where,
ϵ is an integer to shift the starting index
n is the index and counts from $n = 0 \ldots N-1$

4. Sort $\underline{\mathbf{v}(\mathbf{n})}$ in ascending order. Denote this sorted vector $\underline{\mathbf{v}'(\mathbf{n})}$.

5. Determine the interleaver matrix $\underline{\underline{\mathbf{P}}}$ from knowledge of how the elements of $\underline{\mathbf{v}'(\mathbf{n})}$ is permuted to form $\underline{\mathbf{v}(\mathbf{n})}$. The vector $\underline{\mathbf{c}}$ should be interleaved in the same way to form $\underline{\mathbf{c}'}$.

It has been proposed by [S. Crozier, 1999] that ϵ for Turbo codes is usually set to 0, m is either 1 or 2. The value of r could also take into account the code repetition rate, so that r would equal the amount of repetitions of each bit which is sent. If the code is non-repetitive then $r = 1$.

Design example
To illustrate the permutation method of the golden interleaver, an example is established using:

$$\begin{aligned} m &= 2 \\ N &= 6 \\ r &= 1 \\ \psi &= 3 \bmod 1 = 0 \\ \epsilon &= 0 \end{aligned}$$

Thus,

$$\begin{aligned} b = \mathrm{N} \cdot \frac{(a^m + \psi)}{r} &= 6 \cdot \frac{(0.6180^2 + 0)}{1} = 2.2915 \\ \underline{\mathbf{v}(\mathbf{n})} &= 0 + n \cdot 2.2915 \bmod 6 \\ &= \begin{bmatrix} 0 & 2.2915 & 4.5831 & 0.8746 & 3.1662 & 5.4577 \end{bmatrix} \end{aligned}$$

After this, the vector must be sorted in ascending order, which could be carried out by the quicksort sorting method or any other sorting method with various degrees of complexity [Loudon, 1999, p. 308].

This would yield the sorted vector $\underline{\mathbf{v}'(\mathbf{n})}$:

$$\underline{\mathbf{v}'(\mathbf{n})} = \begin{bmatrix} 0 & 0.8746 & 2.2915 & 3.1662 & 4.5831 & 5.4577 \end{bmatrix}$$

In other words to determine how to actually permute an incoming vector $\underline{\mathbf{c}}$ to form $\underline{\mathbf{c}'}$, the permutation matrix $\underline{\underline{\mathbf{P}}}$ can be established as:

$$\underline{\underline{\mathbf{P}}} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Where it is seen that all indices, except the first and last, are mapped to different indices.

3.2.2 S-random interleaver

The S-random type of interleaver is introduced as a method to randomize the amount of permutation and is a rather heuristic approach to permute a vector of data. This interleaver design proposed by [Divsalar and Pollara, 1995] is of interest, because it is specifically designed to increase separation between 1's in low weight code words when these are permuted using a specific spreading constraint, S.

Two algorithms, each leading to the permuted code vector, are described. Both methods rely on selecting random indices, and testing if the distance to the previous index is larger than S. The first method retries discarded indices randomly, whereas the second method tries to input disregarded indices as soon a possible. The second method will not always converge to a valid solution.

The algorithm of designing the method of interleaving, can be described by the following steps:

1. Let $\underline{\mathbf{c}}$ be the code vector of length N. Indices are $i = 0 \ldots N-1$.

2. Choose the constant S to that:

$$S \leq \sqrt{\frac{N}{2}} \tag{3.8}$$

   which will ensure that the first method always converges to a valid solution.

3. Select a random index i from $\underline{\mathbf{c}}$, and determine:

$$d = |i - j| \tag{3.9}$$

   Where,
   j is the last index in the permuted vector $\underline{\mathbf{c}'}$ at this point ($\underline{\mathbf{c}'}$ is initially empty).

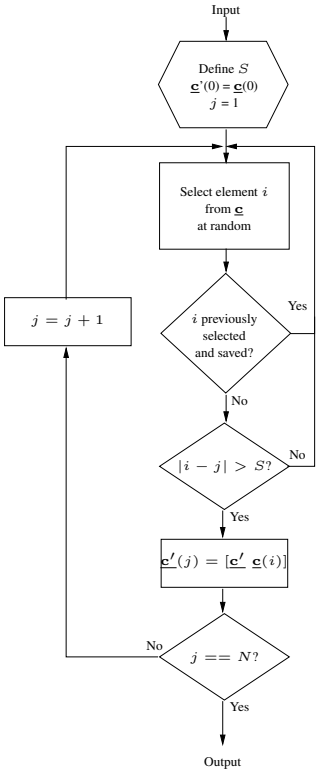


Figure 3.1: S-random method 1 illustrated as a flowchart

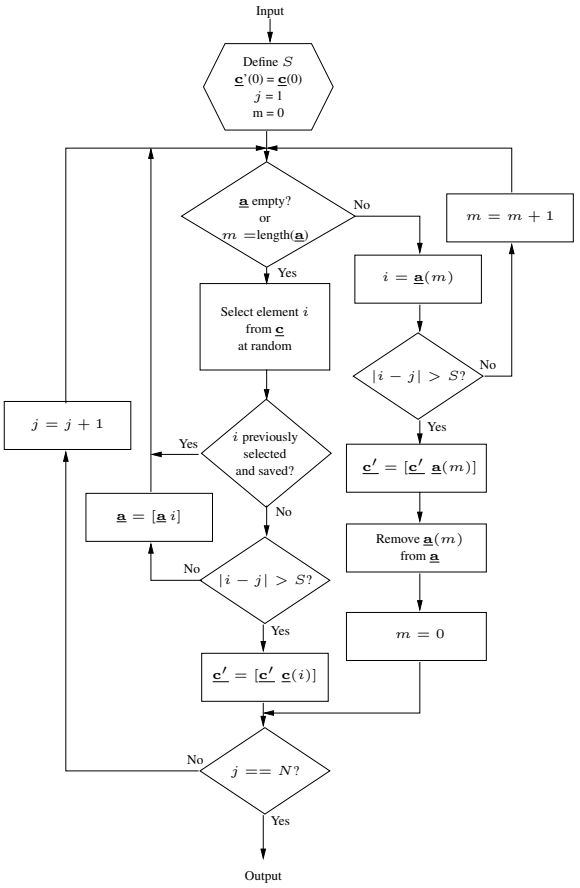


Figure 3.2: S-random method 2 illustrated as a flowchart

4. If $d > S$, store i at index $j + 1$ in $\underline{\mathbf{c}}'$.

Method 1:

5.1. If $d \leq S$, discard i.

Method 2:

5.2.a. If $d \leq S$, store i in the vector of discarded indices, $\underline{\mathbf{a}}$.

5.2.b. If the vector $\underline{\mathbf{a}}$ contains any integers, retry step 3 for each of the values in $\underline{\mathbf{a}}$ using the FIFO principle, before retrying with untested, new, randomly selected i's.

6. Repeat from step 3 until $\underline{\mathbf{c}}'$ contains N integers.

The two methods are illustrated in Figure 3.1 and Figure 3.2 respectively.

Design example
To illustrate the permutation, a code word of $N = 6$ is introduced. The index i is chosen: $1 \rightarrow 2 \rightarrow 4 \rightarrow 5 \rightarrow 3 \rightarrow 0 \rightarrow 2 \rightarrow 5 \rightarrow$. The required spreading is $S = 1$. By using

method 1, the permuted sequence becomes:

$$\underline{\mathbf{c}'_{\mathbf{1}}} = \begin{bmatrix} 1 & 4 & 0 & 2 & 5 & 3 \end{bmatrix}$$

The permutation is therefore done using the interleaver matrix $\underline{\underline{\mathbf{P}}}$:

$$\underline{\underline{\mathbf{P}}} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

Using method 2 on this same sequence of i's results in the sequence:

$$\underline{\mathbf{c}'_{\mathbf{2}}} = \begin{bmatrix} 1 & 4 & 2 & 5 & 3 & 0 \end{bmatrix}$$

As it can be seen, method 2 does not result in much permutation, however, increasing the value of S will increase the spreading, but this might make the method unable to converge to a valid solution.

However, due to the heuristic nature of this interleaving method, it relies on the quality of the random number generator used to select the integers i, which in itself is derived from a probabilistic analysis [Dolinar and Divsalar, 1995].

In order to maintain some of the properties of the S-random interleaver, while maintaining the deterministic nature of the golden interleaver, the quadratic interleaver is introduced.

3.2.3 Quadratic interleaver

It has been proposed that interleavers based on pseudo-random selection of indices, such as the S-random interleaver, are hard to represent in compact form and actually harder to implement than deterministic interleavers [Takeshita and Costello, 2000].

The proposed solution is the quadratic interleaver, which can be determined in closed form. The procedure for permutation is as follows [Takeshita and Costello, 1998]:

1. Let $\underline{\mathbf{c}}$ be the code vector of length N where N is a power of 2. Indices are $i = 0 \ldots N-1$.
2. Choose an odd constant k
3. Compute the vector $\underline{\mathbf{c}'}$:

$$\underline{\mathbf{c}'} = \frac{k \cdot i(i+1)}{2} \bmod N \tag{3.10}$$

This is illustrated by the following example:

Design example
The following values and vectors are chosen for the example:

$$
\begin{aligned}
N &= 8 \\
k &= 23 \\
\underline{\mathbf{c}} &= \begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{bmatrix}
\end{aligned}
$$

Inserting values in Equation 3.10 gives:

$$
\underline{\mathbf{c}}' = \frac{23 \cdot i^2 + i}{2} \bmod 8
$$

Which results in a permutation to:

$$
\underline{\mathbf{c}}' = \begin{bmatrix} 0 & 7 & 5 & 2 & 6 & 1 & 3 & 4 \end{bmatrix}
$$

Again, this is described by $\underline{\underline{\mathbf{P}}}$:

$$
\underline{\underline{\mathbf{P}}} = \begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0
\end{bmatrix}
$$

As it can be seen in this example the first element of $\underline{\mathbf{c}}'$ never changes position. This is an undesired property of the quadratic interleaver, but it is a general property of the deterministic interleaver designs discussed in this chapter.

3.3 Comparative simulations

From the analysis of three methods of permuting integers, it appears that any method is equally well suited for the establishment of interleavers. However, as mentioned in Section 3.1, the interleaver methods must comply with two different constraints. Firstly, the interleaving must happen in a way that seems as random as possible. Furthermore the interleaver methods must ensure a relatively large distance between the placement of an element in the input vector and the placement of the same element in the output vector.

The evaluation of the interleaver quality regarding these two constraints are therefore evaluated in order to conclude on the best choice. The initial examination is performed using scatter diagrams, where the index i appears as the horizontal axis and the re-mapped index j appears at the vertical

axis. If the goal of random shifts is fulfilled, the scatter diagrams will look random with no discernible pattern.

In order to compare how well each method succeeds in making the distance between i and j as large as possible, the evaluation of the distance between i and j is determined by the average norm-1 distance for all indices:

$$\tilde{\Delta}_{i,j} = \frac{1}{N} \sum_{k=0}^{N-1} |i_k - j_k| \tag{3.11}$$

From the scatter diagrams, the randomness of the various methods can be visually inspected, but in practice it is very hard to ensure that no patterns are present by using simple visual inspection. Therefore, a statistical approach is used to ensure randomness, namely the runs test used on the dispersion between all pairs of i and j.

The runs test is therefore described in the next section.

3.3.1 Runs test

In this test it is to be determined if a certain parameter changes from time to time, and if this deviation is random or not. While it is possible to estimate and test the randomness relative to any statistical parameter [Bendat and Piersol, 1986, p. 344], it has been chosen to only look at the changes in mean.
Thus, the number of groups of consecutive values above and below the mean value must be identified and classified. For example, for an interleaver of length $N = 10$, if "1" denotes a value higher and a "0" lower than the mean value of the dataset, the dataset relative to the mean can be classified as:

$$\gamma(|i-j|) = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 \end{bmatrix} \tag{3.12}$$

Let n denote the amount of 1's, m denote the amount of 0's, and r is the number of consecutive values of "1" and "0" for γ. For the example in Equation 3.12, this yields:

$$\begin{aligned} n &= 4 \\ m &= 6 \\ r &= 6 \end{aligned}$$

In order to test the hypothesis that the interleaver is random, a probability distribution function must be defined and the p-value compared to the significance level. It can be shown that the probability mass function can be described by [Gibbons and Chakraborti, 2003, p. 80]:

$$f_R(r) = \begin{cases} \dfrac{2 \cdot \binom{n-1}{\frac{r}{2}-1}\binom{m-1}{\frac{r}{2}-1}}{\binom{n+m}{n}} & r \text{ is even} \\ \dfrac{\binom{n-1}{\frac{r-1}{2}}\binom{m-1}{\frac{r-3}{2}} + \binom{n-1}{\frac{r-3}{2}}\binom{m-1}{\frac{r-1}{2}}}{\binom{n+m}{n}} & r \text{ is odd} \\ & \text{for } r = 2 \ldots k, \text{ where } k = \text{number of runs.} \end{cases} \tag{3.13}$$

The p-value is determined by a rewritten notation of [Ross, 2004, p. 534]:

$$\text{p-value} = 2 \cdot \min\left(f_R\{r \geq R\}, f_R\{r \leq R\}\right) \tag{3.14}$$

On this basis, the interleaving methods are now compared, and the runs test carried out on the absolute difference of the indexing ($|i - j|$) relative to the mean value of $|i - j|$. In other words, the randomness of dispersion is investigated.

3.3.2 Results

The block length N is set to 1024 in all simulations described here, otherwise the values of all variables are reused from the examples in this chapter.

The results from the initial examinations can be seen in Figure 3.3 to Figure 3.6.

For the S-random interleaver, $S = 22$ for method 1 and 2. As seen from Figure 3.5, relatively few remappings are completed, since the randomness of the choice of index makes permutation less likely than if the indices were selected in ascending order. In order to improve on the spreading, the value of S should be increased. By selecting $S = N/3$, the improved spreading in Figure 3.7 is achieved.

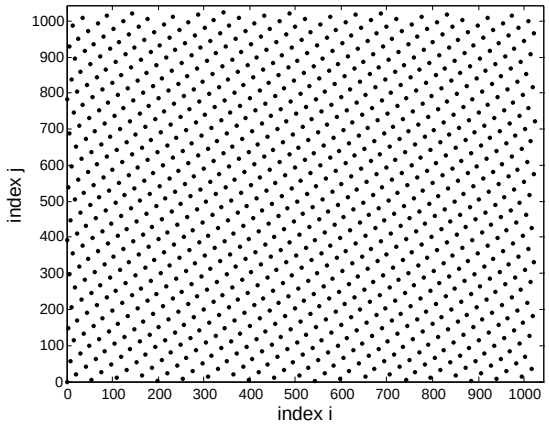


Figure 3.3: Scatter diagram of permutation by golden interleaver.

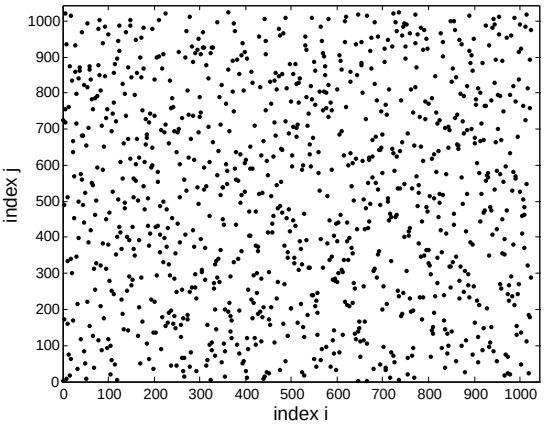


Figure 3.4: Scatter diagram of permutation by S-random interleaver (method 1), S=22, 4 iterations.

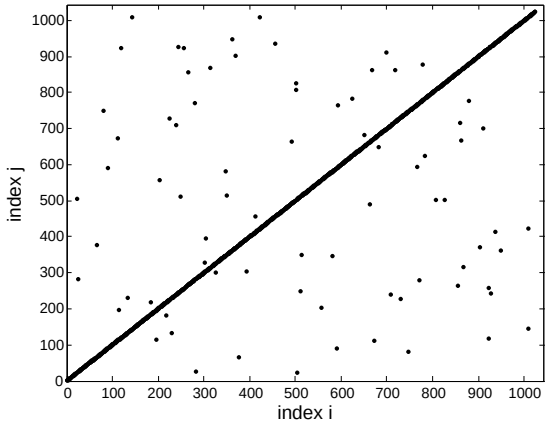


Figure 3.5: Scatter diagram of permutation by S-random interleaver (method 2), S=22, 1 iteration.

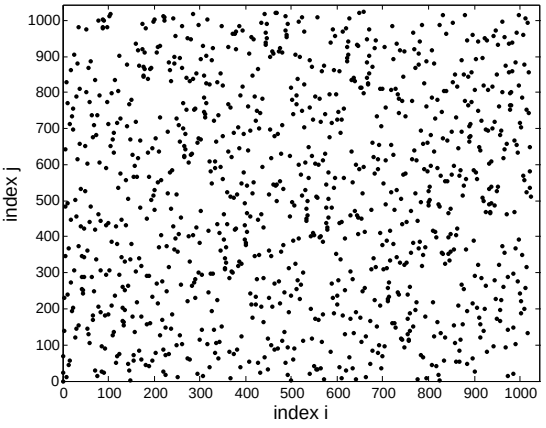


Figure 3.6: Scatter diagram of permutation by quadratic interleaver.

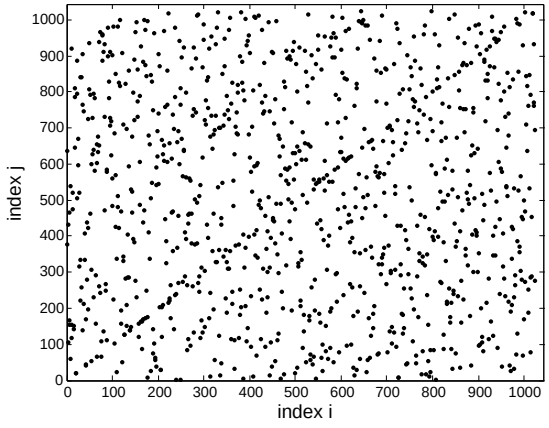


Figure 3.7: Scatter diagram of permutation by S-random interleaver (method 2), S=341, 1 iteration.

In order to test the compliance of the second requirement, Equation 3.11 is evaluated for all methods:

$$\text{Golden interleaver: } \tilde{\Delta}_{i,j} = 340.6113 \text{ indices}$$
$$\text{S-random interleaver (Method 1, S=22): } \tilde{\Delta}_{i,j} = 338.8652 \text{ indices}$$
$$\text{S-random interleaver (Method 2, S=22): } \tilde{\Delta}_{i,j} = 26.9336 \text{ indices}$$
$$\text{S-random interleaver (Method 2, S=341): } \tilde{\Delta}_{i,j} = 333.3379 \text{ indices}$$
$$\text{Quadratic interleaver: } \tilde{\Delta}_{i,j} = 331.8008 \text{ indices}$$

Interleaving method	**P-value**	**Dispersion**
Golden	0.0000	Non-random
S-random (method 1, S=22)	0.8011	Random
S-random (method 2, S=22)	0.0000	Non-random
S-random (method 2, S=341)	0.8038	Random
Quadratic	0.3628	Random

Table 3.1: Results for the runs test for randomness of the various described interleaving methods.

Finally, the runs test is carried out for all methods above. The results are presented in Table 3.1 for a 5% level of significance, which means that if a method is deemed to result in a non-random sequence it happens with 95% certainty.

3.4 Intermediate conclusion

Three entirely different methods of designing an interleaving pattern has been analysed and described. It has been determined that an interleaver should interleave the data in a random fashion while still ensuring that the difference between the placement of an element before and after the interleaving is high.

The golden interleaver is disregarded, since it shows a systematic pattern, which indicates that $\tilde{\Delta}_{i,j}$ has low variance between different groups of i and j, also, this is in accordance with the runs test result.

The second method of S-random interleaving for $S = 22$ shows an even stronger pattern, indicating that only few indices are permuted. However, by increasing the amount of required spread, it does appear that this increases randomness in the permutation, albeit with a slight trend left. The result of the runs test shows that this method with the increased required spread is outputting a random pattern.

The S-random interleaver (method 1) and quadratic interleaver have nearly identical scatter plots which both look randomly distributed. The runs test suggests with a higher significance level that the S-random interleaver using method 1 has more random dispersion than the quadratic interleaver. However, due to the nature of the simplicity of deterministic interleavers, the quadratic interleaver is chosen as the interleaver for implementation in Turbo encoding. This sets the demand that the block size must be a power of 2.

The interleaver function has now been analysed, and it has been determined that the quadratic interleaver is the most suitable of the analysed algorithms. The exploration of the algorithmic domain in the modified A^3 model described in Section 1.2 has been done. The next step is therefore to analyse the decoding functionality and find the algorithms to use for this. The analysis of the decoding function will be done in the next chapter.

4

Turbo decoder

As described in Chapter 1, the concept of Turbo Coding (TC) involves using more than one encoder on the same input sequence, and then decoding in an iterative manner where information obtained from decoding one code is used to decode the others.

The purpose of this chapter is to analyse and discuss the decoding function in the TC scheme.

In Chapter 2, it was determined that only the Parallel Concatenated Convolutional Code (PCCC) type of turbo encoder structure is considered in this project. Therefore, no other decoder structures than the one used for decoding this type of TC will be considered. Some additional assumptions are considered in order to describe the theory of the turbo decoder.

4.1 Assumptions

It is assumed that the encoder structure is a parallel concatenation of two Recursive Systematic Convolutional (RSC) encoders as described in Chapter 2, and that the RSC encoders incorporate trellis termination and the modulated signal is transmitted over an AWGN channel as discussed in Section 1.3.1. Furthermore it is assumed that the output from the demodulator is normally distributed around -1 if a 0 has been sent and around 1 if a 1 has been sent, which is the case for the BPSK modulation scheme. It is furthermore assumed that each bit from the demodulator is independent of previous outputs.

4.2 Soft-input and soft-output

As previously described, the concept of TC consists of using more than one code on the same input and use knowledge obtained from decoding one code when decoding the others. In order to do this a special type of decoders must be employed. Each decoder must be able to include some extra knowledge in the decoding process and be able to provide such information for the other decoder.

Such decoders are called Soft-Input Soft-Output (SISO) decoders. This means that they accept soft input which is a number that represents both the value of the bit and the certainty. For instance an input can be -0.8 which means that the bit is zero with 80 % probability. A SISO should also output this type of information.

As it can be seen in Figure 4.1, a SISO decoder for use in a turbo decoder structure must accept the a priori values $L(\underline{\mathbf{x}})$ of all bits as input. It should also accept the values $\underline{\mathbf{y}}$ from the demodulator [M. R. Soleymani, 2002, p. 29]. The output from such a decoder must be the extrinsic information (information contained in the redundancy added by the encoder) $L_e(\underline{\hat{\mathbf{x}}})$ and the a posteriori values $L(\underline{\hat{\mathbf{x}}})$.

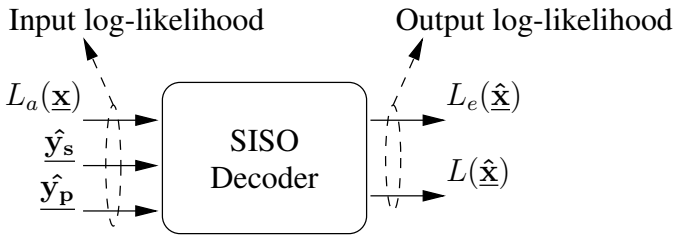


Figure 4.1: SISO decoder. [M. R. Soleymani, 2002, Fig. 2.1].

The operation of the decoder depicted above can mathematically be described by the equation below, assuming that the coding scheme is systematic:

$$L(\underline{\hat{\mathbf{x}}}) = a\underline{\hat{\mathbf{y}}_\mathbf{s}} + L_a(\underline{\mathbf{x}}) + L_e(\underline{\hat{\mathbf{x}}}) \tag{4.1}$$

Where

$L(\underline{\hat{\mathbf{x}}})$	is the Log Likelihood Ratio (LLR) which is output from the encoder.
$\underline{\hat{\mathbf{y}}_\mathbf{s}}$	is the systematic part of the received bits.
$L_e(\underline{\hat{\mathbf{x}}})$	is the extrinsic LLR.
$L_a(\underline{\mathbf{x}})$	is the LLR calculated from the a priori probabilities of the bits being sent.
a	is a constant dependent on the type of decoder.

The three parts for estimating the LLR of the information bits are independent [Sklar, 2002, p. 5].

4.3 Iterative decoding

An iterative decoder with two SISO decoders is depicted in Figure 4.2.

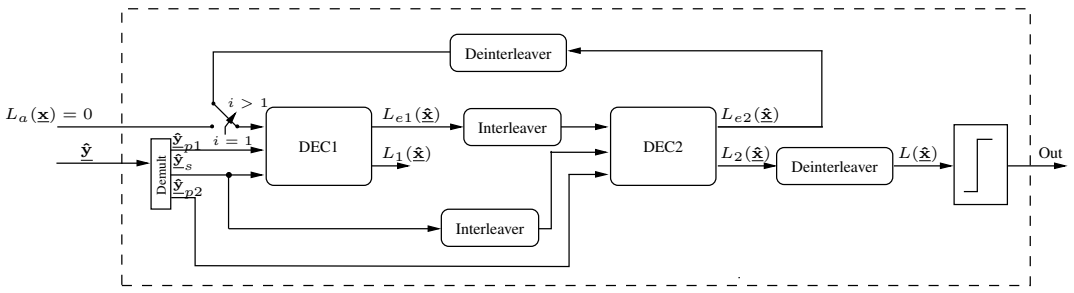


Figure 4.2: Iterative decoder with two SISO decoders. [M. R. Soleymani, 2002, Fig. 2.2].

As it can be seen in Figure 4.2, the decoders take the extrinsic information from the other decoder as their a priori value, which helps the overall decoder iterate closer to the correct result. In the first iteration, the first decoder DEC1 has no extrinsic information from DEC2, so to avoid biasing the first iteration, $L_a(\underline{\mathbf{x}})$ is set to $\underline{\mathbf{0}}$. In the first iteration DEC1 calculates the extrinsic information as follows:

$$L_{e1}(\underline{\hat{\mathbf{x}}}) = L_1(\underline{\hat{\mathbf{x}}}) - \left[a\underline{\hat{\mathbf{y}}} + L_a(\underline{\mathbf{x}})\right] \tag{4.2}$$
$$L_{e1}(\underline{\hat{\mathbf{x}}}) = L_1(\underline{\hat{\mathbf{x}}}) - a\underline{\hat{\mathbf{y}}} \tag{4.3}$$

The extrinsic information from DEC1 is interleaved and passed to DEC2 and taken in as the a priori values. Extrinsic information is now generated in DEC2 for the next iteration:

$$L_{e2}(\underline{\hat{\mathbf{x}}}) = L_2(\underline{\hat{\mathbf{x}}}) - \left[a\underline{\hat{\mathbf{y}}} + L_{e1}(\underline{\hat{\mathbf{x}}})\right] \tag{4.4}$$

Where it is deinterleaved and passed as a priori values to DEC1 which then generates extrinsic information like this:

$$L_{e1} = L_1(\underline{\hat{\mathbf{x}}}) - a\underline{\hat{\mathbf{y}}} - L_a(\underline{\mathbf{x}}) \tag{4.5}$$

This loop runs either for a fixed number of iterations or until the extrinsic value has settled i.e. only varies within some tolerance level. The a posteriori values for the information bits from DEC2 become:

$$L_2(\underline{\hat{\mathbf{x}}}) = a\underline{\hat{\mathbf{y}}} + L_{e1}(\underline{\hat{\mathbf{x}}}) + L_{e2}(\underline{\hat{\mathbf{x}}}) \tag{4.6}$$

Which is deinterleaved and then the hard output (ones and zeros) is formed.

The iterative decoder structure based on two SISO decoders has now been explained, and the next section will focus on choosing and describing a SISO decoder.

4.4 Choosing a decoding algorithm

As stated in [Liang et al., 2004, p. 2] there are many possible algorithms that can be used for decoding in a TC scheme, and some of the optimal algorithms are MAP and Log-MAP. The MAP is a *maximum a posteriori* rule that has been implemented in, for instance, the BCJR algorithm by [Bahl et al., 1974]. It is, however, difficult to use in practice as probabilities are not well represented numerically and the algorithm requires solving many non-linear functions and many other calculations [M. R. Soleymani, 2002, p. 30]. The problem of numerical representation has been diminished in the Log-MAP algorithm, but unfortunately it is still an algorithm of high computational complexity [M. R. Soleymani, 2002].

If for example restricted by processing capabilities there are alternatives to the optimal algorithms such as the suboptimal algorithms Max-Log-MAP and Soft Output Viterbi Algorithm (SOVA).

The Max-Log-MAP employs some approximations to decrease the complexity of Log-MAP, and SOVA gives a soft-output based on the Viterbi algorithm for finding the most probable path (sequence) [M. R. Soleymani, 2002, p. 30]. SOVA has much reduced complexity, but e.g. at a BER of 10^{-4} it performs approximately 0.7 dB worse than the optimal MAP algorithm [M. R. Soleymani, 2002, p. 30].

As an example the 3rd Generation Partnership Project (3GPP) standard has introduced TC with both Max-log-MAP and SOVA decoders for different applications [Chaikalis et al., 2002]. They state that the Log-MAP is well-suited for non-real-time applications, such as file transfer, where a high BER is demanded to keep the throughput high while latency is not an issue. On the other hand, they state that in real-time applications (e.g. video streaming) where the demand is for low latency and less for throughput, SOVA is the preferred one.

The focus of this project is not to make the absolutely best performing system in terms of error correction, but to construct a system that performs relatively well and to show how a soft-core processor will handle such coding schemes compared to a soft-core implementation aided by hardware. Due to the low complexity SOVA is selected for further analysis and implementation.

4.5 Soft Output Viterbi Algorithm

The rest of this chapter is written on the basis of knowledge obtained from [Vucetic and Yuan, 2000, p. 117-137 & p. 171-182].

A decoder for a Turbo Coding (TC) scheme must comply with the following three demands:

- The decoder must be able to decode convolutional codes (either RSC or others depending on the encoding scheme).
- The decoder must accept soft-input and output soft information in form of both a normal Log Likelihood Ratio (LLR) and an extrinsic LLR (as previously described).
- The decoder must take a priori probabilities into account when decoding the information.

All these properties are present in a decoder based on the Soft Output Viterbi Algorithm (SOVA) algorithm. In order to best describe the theory behind this decoder, the Viterbi Algorithm (VA) is described first because it contains two of the three properties above, and the SOVA is a rather simple extension of this.

4.5.1 Viterbi Algorithm

The basic concept behind the VA is to look at the decoding process as a problem of estimating which states the encoder has visited, based on the received information. It is assumed, as previously mentioned, that the decoder outputs soft information and that the output of the demodulator is normally distributed with variance σ and mean -1 if a zero has been sent and mean 1 if a one has been sent.

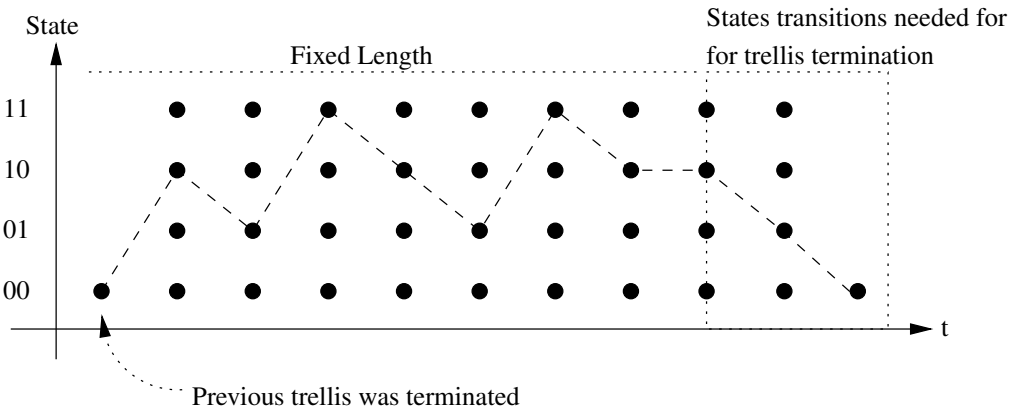


Figure 4.3: Grid in which all the possible state-sequences in the encoder can be illustrated. The dashed line illustrates a possible path.

In order to estimate the state-sequence in the encoder that most likely has generated the sequence that has been received, it is necessary to view the set of possible state-sequences as a grid as illustrated in Figure 4.3.

As it can be seen on Figure 4.3, it is assumed that the amount of information bits are fixed and that the encoder terminates the trellis after each sent block (i.e. the encoder sends the bits needed for it to get to the zero-state. These bits contain no relevant information other than the fact that it ensures the zero state in both ends of the block. The termination bits are therefore not necessarily sent through the channel.

A branch metric (or transition cost) is associated with each transition between points in the trellis grid. This means that each transition is associated with some cost, where a likely transition would yield a low cost and a unlikely transition would yield a high cost. Therefore, the problem of finding the most probable state sequence is converted to a problem of finding the path through the grid which is associated with the lowest cost. Such a branch metric is derived in the following section.

Deriving the Branch Metric

In this derivation, and the rest of the report, the following variables will be used:

- $\underline{\mathbf{x}}$ - A sequence of input-bits to the encoder.
- x_t - The t'th element of the sequence $\underline{\mathbf{x}}$.
- $\underline{\mathbf{y}}$ - A sequence of bit-strings, each corresponding to a single input bit to the encoder.
- y_t - The bit-string corresponding to the t'th input to the encoder (length n).
- $y_{t,i}$ - The i'th element of the t'th bit-string in $\underline{\mathbf{y}}$.
- $\underline{\hat{\mathbf{y}}}$ - The received sequence of (soft-information) bit-strings.
- $\hat{y}_t$ - The t'th received bit-string.
- $\hat{y}_{t,i}$ - The i'th element of the t'th received bit-string.
- $\underline{\hat{\mathbf{x}}}$ - The output-stream of the decoder (which is an estimate of $\underline{\mathbf{x}}$).
- $\hat{x}_t$ - The t'th element of the estimate of $\underline{\mathbf{x}}$.

The purpose of the branch metric is, as described above, to associate a cost with each state-transition. It is therefore natural to start by finding the sequence $\underline{\hat{\mathbf{x}}}$ that is the most likely to have been encoded to generate the sequence $\underline{\mathbf{y}}$, which has been sent over the channel to form $\underline{\hat{\mathbf{y}}}$. A

derivation then results in an expression that can be split up into a cost for each transition.

In order to decode, it is necessary to find the sequence $\underline{\hat{\mathbf{x}}}$ that is the most probable, given the received sequence $\underline{\hat{\mathbf{y}}}$:

$$\max_{\underline{\hat{\mathbf{x}}}} \mathrm{P}(\underline{\mathbf{x}} = \underline{\hat{\mathbf{x}}} | \underline{\hat{\mathbf{y}}}) = \max_{\underline{\hat{\mathbf{x}}}} \frac{\mathrm{P}(\underline{\mathbf{x}} = \underline{\hat{\mathbf{x}}}) \cdot \mathrm{p}(\underline{\hat{\mathbf{y}}} | \underline{\mathbf{x}} = \underline{\hat{\mathbf{x}}})}{\mathrm{P}(\underline{\hat{\mathbf{y}}})} \tag{4.7}$$

In this maximization, the term $\mathrm{P}(\underline{\hat{\mathbf{y}}})$ is just a constant and is therefore removable which leaves:

$$\max_{\underline{\hat{\mathbf{x}}}} \mathrm{P}(\underline{\mathbf{x}} = \underline{\hat{\mathbf{x}}}) \cdot \mathrm{p}(\underline{\hat{\mathbf{y}}} | \underline{\mathbf{x}} = \underline{\hat{\mathbf{x}}}) \tag{4.8}$$

The terms $\mathrm{P}(\underline{\mathbf{x}} = \underline{\hat{\mathbf{x}}})$ and $\mathrm{p}(\underline{\hat{\mathbf{y}}} | \underline{\mathbf{x}} = \underline{\hat{\mathbf{x}}})$ is in fact, because of the independence of each element of $\underline{\mathbf{x}}$, the products of several probabilities, like this:

$$\mathrm{P}(\underline{\mathbf{x}} = \underline{\hat{\mathbf{x}}}) = \prod_{t=1}^{\tau} \mathrm{P}(x_t = \hat{x}_t) \tag{4.9}$$

$$\mathrm{p}(\underline{\hat{\mathbf{y}}} | \underline{\mathbf{x}} = \underline{\hat{\mathbf{x}}}) = \prod_{t=1}^{\tau} \prod_{i=1}^{n} \mathrm{p}(\hat{y}_{t,i} | x_t = \hat{x}_t) \tag{4.10}$$

Since there is a relation between x_t and y_t and it is assumed that the output of the demodulator is normally distributed with mean -1 if $x_t = 0$ and 1 if $x_t = 1$ and variance σ it is possible to rewrite $\mathrm{p}(\hat{y}_{t,i} | x_t = \hat{x}_t)$ to:

$$\mathrm{p}(\hat{y}_{t,i} | x_t = \hat{x}_t) = \mathrm{p}(\hat{y}_{t,i} | y_{t,i}) = \frac{1}{\sqrt{2\pi \cdot \sigma^2}} e^{\frac{-(\hat{y}_{t,i} - y_{t,i})^2}{2 \cdot \sigma^2}} \tag{4.11}$$

By using this and by taking the natural logarithm of Equation 4.9 (this has no effect on which $\underline{\hat{\mathbf{x}}}$ solves the maximization) the following is found:

$$\begin{aligned} \max_{\underline{\hat{\mathbf{x}}}} \ln(\mathrm{P}(\underline{\mathbf{x}} = \underline{\hat{\mathbf{x}}}) \mathrm{p}(\underline{\hat{\mathbf{y}}} | \underline{\mathbf{x}} = \underline{\hat{\mathbf{x}}})) &= \max_{\underline{\hat{\mathbf{x}}}} \sum_{t=1}^{\tau} \ln(\mathrm{P}(x_t = \hat{x}_t)) \\ &+ \sum_{i=1}^{n} \ln(1) - \frac{\ln(2\pi \cdot \sigma^2)}{2} - \frac{(\hat{y}_{t,i} - y_{t,i})^2}{2 \cdot \sigma^2} \qquad (4.12) \\ &\sim \max_{\underline{\hat{\mathbf{x}}}} \sum_{t=1}^{\tau} \ln(\mathrm{P}(x_t = \hat{x}_t)) - \sum_{t=1}^{\tau} \sum_{i=0}^{n} (\hat{y}_{t,i} - y_{t,i})^2 \qquad (4.13) \end{aligned}$$

The constant terms are removed from Equation (4.12) because they have no effect on which $\underline{\hat{\mathbf{x}}}$ maximizes the expression. The result of this is stated in Equation (4.13). It can be seen that the maximization in Equation 4.13 is equivalent to the following:

$$\min_{\underline{\hat{\mathbf{x}}}} \sum_{t=1}^{\tau} \sum_{i=1}^{n} (\hat{y}_{t,i} - y_{t,i})^2 - \ln(\mathrm{P}(x_t = \hat{x}_t)) \tag{4.14}$$

This expression can, in fact, be split up into a term for each state transition (by splitting the sum over t). Furthermore it can be seen that the first term of this expression looks just like a Euclidean distance (with the square root left out) between the received sequence and the sequence that is sent. This similarity ensures that if the output, corresponding to a transition that does not match the transition that generated the received sequence, then the cost will be large.

The branch metric (cost of a specific transition) is therefore defined like this:

$$\nu(t, S, \hat{x}_t) = \sum_{i=1}^{n} (\hat{y}_{t,i} - y_{S,\hat{x}_t,i})^2 - \ln(\mathrm{P}(x_t = \hat{x}_t)) \tag{4.15}$$

Where

S	is the state from which the transition is going
$\hat{x}_t$	is the input to the encoder if it is to make the transition
$\hat{y}_{t,i}$	is the i'th bit in the t'th received bit string
$y_{S,\hat{x}_t,i}$	is the i'th bit which is outputted if the encoder is getting $\hat{x}_t$ is input while being in state S

As described in Section 4.2, the a priori probabilities are sent to the SOVA decoder in form of a log-likelihood. In order to use this in the SOVA it is necessary to use the following two equations:

$$\mathrm{P}(x_t = 0) = \frac{1}{1 + e^{L_a(x_t)}} \tag{4.16}$$

$$\mathrm{P}(x_t = 1) = \frac{e^{L_a(x_t)}}{1 + e^{L_a(x_t)}} \tag{4.17}$$

In order to be able to define a distinct path through the trellis, it is necessary to define the path vector consisting of the input sequence, corresponding to the path, and the vector containing the state sequence of this path:

$$\underline{\mathbf{P}} = [\hat{x}_k] \quad \text{where} \quad k \in [0, t] \tag{4.18}$$

$$\underline{\mathbf{S}}(\underline{\mathbf{P}}) = [S_k] \quad \text{where} \quad k \in [0, t] \tag{4.19}$$

It is also possible to define a path metric (i.e. the total cost of a path through the trellis up to the t'th state):

$$\mu_t(\underline{\mathbf{P}}) = \nu(t-1, S_{t-1}(\underline{\mathbf{P}}), P_t) + \mu_{t-1}(\underline{\mathbf{P}}) \tag{4.20}$$

Where

$S_{t-1}(\underline{\mathbf{P}})$	is the state that the path described by $\underline{\mathbf{P}}$ is in at $t-1$
P_t	is the t'th element of the vector $\underline{\mathbf{P}}$
$\mu_{t-1}(\underline{\mathbf{P}})$	is the cost of the path described by $\underline{\mathbf{P}}$ at $t-1$

Finding the most probable transition sequence

The VA uses the following knowledge in order to find the most probable path through the trellis:

- The encoder uses trellis termination, which means that the trellis for each block must start and end in state zero. (This is depicted on Figure 4.3).
- The encoder is dividing the input sequence into blocks of fixed length, which means that the part of the received bit-sequence that contains relevant information is of fixed length.
- Because of the fact that the encoder is binary, it only encodes one bit at a time. This, in turn, means that there can only be two inputs into each node, and from each node only two outputs.
- For each state S_k and t there is only one best path from state zero at $t = 0$ to state S_k at time t.

By use of this knowledge, it is possible to find the path with the lowest cost using the following steps:

1. Start at $t = 1$ and set $\mu_0(S = S_0) = 0$ and $\mu_0(S \neq S_0) = \infty$ (that is, cost of starting in state S_0 is zero and the cost of starting in any other state is infinitely high. This ensures that the found path starts in state 0).
2. Calculate the branch metrics for all branches entering a node at t as:

$$\nu(t, S, \hat{x}_t) = \sum_{i=1}^{n} (\hat{y}_{t,i} - y_{S,\hat{x}_t,i})^2 - \ln(\mathrm{P}(x_t = \hat{x}_t)) \quad \text{for all possible } \hat{x}_t \text{ and } S \tag{4.21}$$

3. Calculate the path metric for all branches entering all nodes at t by adding each branch metric to the survivor of the node from which it comes (or either $\mu_0(S = S_0)$ or $\mu_0(S \neq S_0)$).
4. Find the survivor for each node by finding the path entering the node with the lowest path metric.
5. Add one to t and repeat from step 2 if $t = \tau + 1$.
6. When $t = \tau + 1$ then the survivor in state S_0 will be the path with the lowest cost.

Example

This section introduces an example to show the theory applied in practice. All the possible states are seen in Figure 4.4.

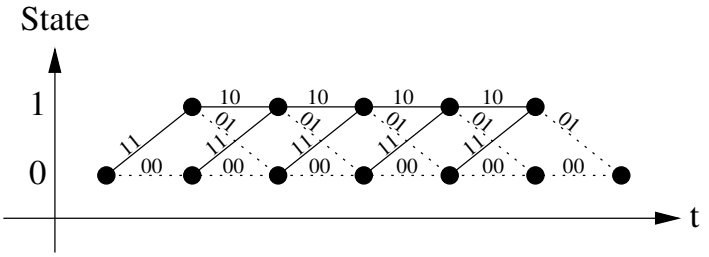


Figure 4.4: Trellis grid showing transmitted ones and zeros as solid and dashed lines respectively.

For this example the received values are:

$$\underline{\hat{\mathbf{y}}} = [\,[0\ 0.8]\ [-0.6\ 0.9]\ [-0.3\ \ -0.8]\ [1.2\ 0.6]\ [0.7\ \ -0.7]\ [-0.9\ 1]\,]$$

At $t = 0$ the a priori probabilities are equal as seen in the equation below, as there are no prior knowledge about the value of the source data.

$$P(x = 0) = 0.5 \quad \text{and} \quad P(x = 1) = 0.5$$

For $t = 1$ the branch costs are:

$$\begin{aligned} v(1, S_0, 0) &= (0+1)^2 + (0.8+1)^2 - \ln(0.5) = 2.33 \\ v(1, S_0, 1) &= (0-1)^2 + (0.8-1)^2 - \ln(0.5) = 1.73 \\ v(1, S_1, 0) &= (0+1)^2 + (0.8-1)^2 - \ln(0.5) = 1.73 \\ v(1, S_1, 1) &= (0-1)^2 + (0.8+1)^2 - \ln(0.5) = 2.33 \end{aligned}$$

The path costs for the paths reaching the states at $t = 1$:

$$\begin{aligned} \mu_1([0]) &= \mu_0(S = S_0) + v(1, S_0, 0) = 0 + 2.33 = 2.33 \\ \mu_1([1]) &= \mu_0(S = S_0) + v(1, S_0, 1) = 0 + 1.73 = 1.73 \end{aligned}$$

There are no possible branches from S_1 at $t = 0$ as the sequences are zero-terminated i.e. they start and end with a 0 being transmitted.

The least path metrics at the current t are:

$$\begin{aligned} \mu_{\min,1,S_0} &= \mu_1([0]) = 2.33 \\ \mu_{\min,1,S_1} &= \mu_1([1]) = 1.73 \end{aligned}$$

This means that the surviving paths are:

$$\text{Survivor}\,(t=1, S_0) = [0]$$
$$\text{Survivor}\,(t=1, S_1) = [1]$$

The costs of all possible state transitions at $t = 2$ are:

$$v\,(2, S_0, 0) = (-0.6+1)^2 + (0.9+1)^2 - \ln(0.5) = 4.46$$
$$v\,(2, S_0, 1) = (-0.6-1)^2 + (0.9-1)^2 - \ln(0.5) = 3.26$$
$$v\,(2, S_1, 0) = (-0.6+1)^2 + (0.9-1)^2 - \ln(0.5) = 0.86$$
$$v\,(2, S_1, 1) = (-0.6-1)^2 + (0.9+1)^2 - \ln(0.5) = 6.86$$

The costs of the paths reaching the states at $t = 2$ are:

$$\mu_2\,([0\ 0]) = \mu_1\,([0]) + v\,(2, S_0, 0) = 2.33 + 4.46 = 6.79$$
$$\mu_2\,([0\ 1]) = \mu_1\,([0]) + v\,(2, S_0, 1) = 2.33 + 3.26 = 5.59$$
$$\mu_2\,([1\ 0]) = \mu_1\,([1]) + v\,(2, S_1, 0) = 1.73 + 0.86 = 2.59$$
$$\mu_2\,([1\ 1]) = \mu_1\,([1]) + v\,(2, S_1, 1) = 1.73 + 6.86 = 8.59$$

The surviving paths are the ones whose path costs are least, so the survivors are:

$$\begin{aligned}\text{Survivor}\,(t=2, S_0) &= [1\ 0]\\ \text{Survivor}\,(t=2, S_1) &= [0\ 1]\end{aligned}$$

The metrics for the succeeding transitions are calculated similarly. The results are shown in the trellis in Figure 4.5, where both the branch costs and the path costs are depicted.

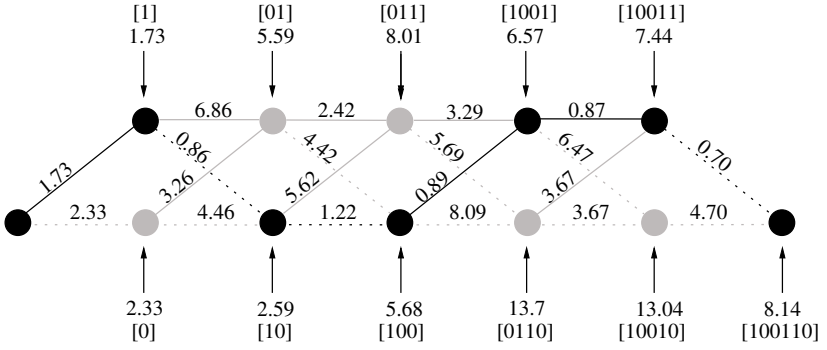


Figure 4.5: Trellis grid with path metrics along the branches.

As it can be seen, the most probable path is $[100110]$.

4.5.2 Soft output in the Viterbi algorithm

In order to make the VA output soft-information, it is necessary to be able to calculate the LLR for each bit in the output stream. The LLR is, as previously described, defined as the base-e logarithm of the probability of $x_t = 1$ given the received sequence divided by the probability of $x_t = 0$ given the received sequence:

$$L_t(\underline{\hat{\mathbf{x}}}) = \ln\left(\frac{\mathrm{P}(x_t = 1|\underline{\hat{\mathbf{y}}})}{\mathrm{P}(x_t = 0|\underline{\hat{\mathbf{y}}})}\right) \tag{4.22}$$

The probability $\mathrm{P}(x_t = 1|\underline{\hat{\mathbf{y}}})$ is, according to [Vucetic and Yuan, 2000, p. 129], proportional to $e^{-\mu_t(\hat{x}_t=1)}$ where $\mu_t(\hat{x}_t = 1)$ is the total cost of the best path with the t'th element being a 1. Similarly the probability $\mathrm{P}(x_t = 0|\underline{\hat{\mathbf{y}}})$ is proportional to $e^{-\mu_t(\hat{x}_t=0)}$. Because of the nature of a ratio, it will not change the LLR to substitute the probabilities with their proportional counterparts. The following derivation is therefore also possible:

$$L_t(\underline{\hat{\mathbf{x}}}) = \ln\left(\frac{e^{-\mu_t(\hat{x}_t=1)}}{e^{-\mu_t(\hat{x}_t=0)}}\right) = \mu_t(\hat{x}_t = 0) - \mu_t(\hat{x}_t = 1) \tag{4.23}$$

From this it can be seen that it is possible to calculate the soft-output of the SOVA decoder by first using the normal VA to find the overall best path, and then to use the VA backwards in the trellis to determine the best path both from and to all nodes in the trellis. The following denotations are used:

$\underline{\mathbf{P}}^0_{\min}$	is the best path in which $\hat{x}_t = 0$
$\underline{\mathbf{P}}^1_{\min}$	is the best path in which $\hat{x}_t = 1$
$\mu_{F,t}(\underline{\mathbf{P}})$	is the cost of the path from state S_0 at $t = 0$ using the path $\underline{\mathbf{P}}$ until t. Calculated during the normal (forward) run of the VA
$\mu_{B,t}(S, \underline{\mathbf{P}})$	is the cost of the path from state S at t to state S_0 at $t = \tau$. Calculated during the backwards run of the VA

Using this, it is possible to use the following expressions for the cost of the minimal paths:

$$\mu_t(\hat{x}_t = 0) = \mu_{F,t}(\underline{\mathbf{P}}^0_{\min}) + \nu(t, S_t(\underline{\mathbf{P}}^0_{\min}), 0) + \mu_{B,t+1}(\underline{\mathbf{P}}^0_{\min}) \tag{4.24}$$

$$\mu_t(\hat{x}_t = 1) = \mu_{F,t}(\underline{\mathbf{P}}^1_{\min}) + \nu(t, S_t(\underline{\mathbf{P}}^1_{\min}), 1) + \mu_{B,t+1}(\underline{\mathbf{P}}^1_{\min}) \tag{4.25}$$

That is, the total cost of a path with $\hat{x}_t = 0$ can be computed by adding the path metric of the survivor in the start node of a transition with $\hat{x}_t = 0$ and add this to the branch metric of the transition and the path metric of the backwards survivor at the end-node. Similarly can be done for the total cost of a path with $\hat{x}_t = 1$

The objective is therefore to find the lowest total path metric with $\hat{x}_t = 0$ and the lowest total path metric with $\hat{x}_t = 1$ and then use Equation 4.23 to calculate the LLR.

4.5.3 Extrinsic output

In Appendix A it has been shown that Equation 4.23 can be rewritten to:

$$L_t(\underline{\hat{\mathbf{x}}}) = 4\hat{y}_{t,1} + L_e(\underline{\hat{\mathbf{x}}}) + L_a(\underline{\mathbf{x}}) \tag{4.26}$$

Where: $L_e(\underline{\hat{\mathbf{x}}})$ is the extrinsic LLR connected to the t'th bit
$\hat{y}_{t,1}$ is the t'th received systematic bit (i.e. the first bit in the t'th bit string).

From this, a very simple expression for the extrinsic information can be found, if the normal LLR is calculated:

$$L_e(\underline{\hat{\mathbf{x}}}) = L(\underline{\hat{\mathbf{x}}}) - 4 \cdot \hat{y}_{t,1} - L_a(\underline{\mathbf{x}}) \tag{4.27}$$

4.6 Intermediate conclusion

The Turbo encoder structure, capable of decoding codes made by a Turbo encoder using the Parallel Concatenated Convolutional Code (PCCC) structure, has been described. The concept of SISO decoders has also been described and the SOVA decoder has been introduced.

This concludes the mapping from functionality to algorithms as shown in the A^3 model from Chapter 1. The next step in the model is to test the compliance of the algorithms with the functional description of TC. This test of compliance will be done in the next chapter, which will also serve as a summation of the algorithms described in this and the previous two chapters.

5

Prototype and compliance test

All the functionalities needed in a Turbo Coding (TC) scheme have been described and analysed, and it has been determined which algorithms can be used to implement these functionalities. The purpose of this chapter is therefore to summarise these algorithms into a prototype and test the compliance of this prototype with the functional description of TC. This compliance test is illustrated on the modified A^3 model as the dashed line from the algorithmic domain to the functional domain (highlighted in Figure 5.1).

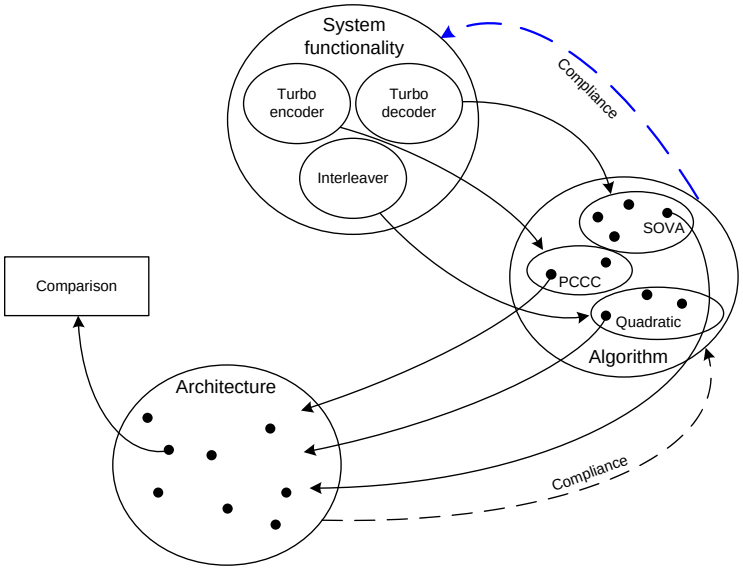


Figure 5.1: The compliance test step of the modified A^3 model.

5.1 Introduction to the prototype

The prototype is coded in MatLAB, as this environment is well-suited for quick simulations of complex systems.

Each of the functionalities described are implemented in their own functions. The interleaver needs a function for permutating data. The decoding is likewise split into two files; one for the iterative decoder structure, and one for the convolutional decoder (Soft Output Viterbi Algorithm (SOVA)).

The structure of the prototype is illustrated in Figure 5.2.

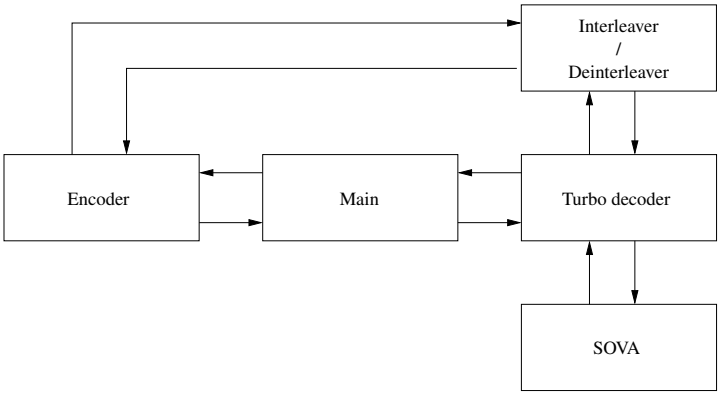


Figure 5.2: The structure of the MatLAB prototype of a Turbo Coding scheme

The encoder is implemented using two Recursive Systematic Convolutional (RSC) encoders with the feedback polynomial $g_1(D) = D^2 + D + 1$ and the output polynomials $g_2(D) = D^2 + 1$ and $g_3(D) = D + 1$. The interleaver design chosen for this prototype is of the quadratic type and the input block length is chosen to be 2048 bits. The decoder is implemented as described in Chapter 4.

The encoder prototype is designed in such a way that other generator polynomials can be tested, even though this is not used in this project. Likewise the possibility of testing different block lengths has been implemented is the prototype.

The Main function, illustrated in the middle of Figure 5.2, generates a random bit string to use as source data. This string is forwarded to the encoder to achieve an encoded bit string which requires the use of the interleaver. However, first all bits are converted from a 0/1 representation to -1/1 to be compatible with the decoder, and afterwards Additive White Gaussian Noise (AWGN) is added to emulate the effect of a noisy channel. After the noise is added, the bit string is passed to the decoder which calls the SOVA decoder, the interleaver and the deinterleaver functions in the needed order.

The number of iterations used to decode the data is fixed in the implementation, so that the test operator can control the number of iterations better than if the number of iteration had been controlled by the magnitude of the change in the extrinsic values.

Channel emulation

As mentioned, the prototype implementation includes an emulation of a channel to be able to test the performance of the system when used with an AWGN channel. Therefore, white Gaussian noise was generated and the power was normalized with respect to the encoded sequence. Afterwards the noise was multiplied by $\sqrt{\frac{1}{10^{SNR/10}}}$, where Signal-to-Noise Ratio (SNR) is the signal-to-noise ratio of the channel emulation in dBs. The noise and the encoded sequence was then added to emulate an AWGN channel.

Test methodology

The purpose of the prototype is to be able to test if the chosen algorithms comply with the functional description of turbo coding. As the purpose of turbo decoding is to use computational power to decrease the Bit Error Rate (BER) by doing more iterations in the turbo decoding structure, it is natural to test if the prototype implementation exhibits this behaviour.

The test consists therefore of encoding long strings of data and then adding noise at different SNRs. The noisy data is then passed to the decoder which performs several iterations before outputting the data.

After the decoding is complete, the decoded data is compared with the original strings, and the BER is calculated.

It is chosen to test the prototype at SNRs ranging from -8 to 2 dB in steps of 1 dB to show the asymptotical behaviour at lower SNRs while still showing the decaying exponential shape for higher SNRs. The test was performed for multiple numbers of iterations ranging from 1 to 8 iterations, since the benefit of performing more iterations is negligible. It is furthermore chosen to repeat the test a 1000 times for each SNR and calculate the mean of the resulting data set to increase the reliability of the results.

5.2 Results

The result of the compliance test is shown in Figure 5.3. It can be seen from the results that more iterations reduce the BER as expected. It is furthermore visible that the decoder does not improve much on the quality of the received data if the SNR is below -5 dB.

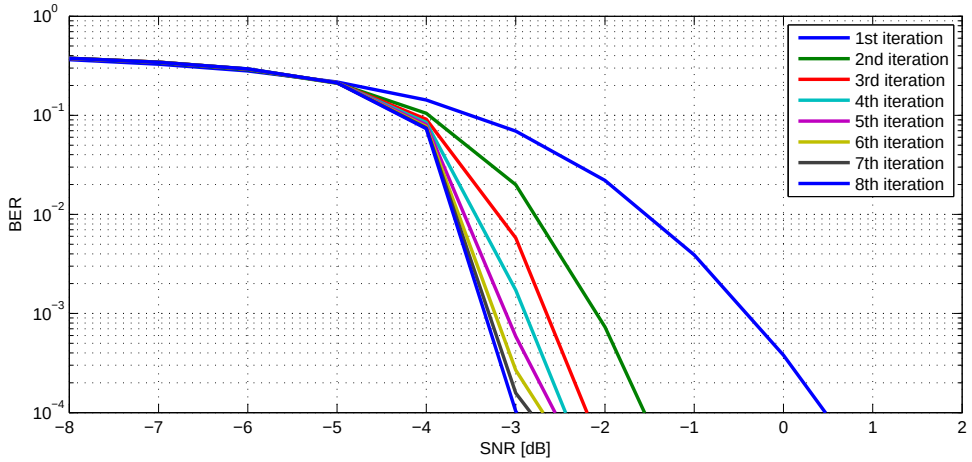


Figure 5.3: Simulation results from MatLAB. Each curve shows the result of a certain number of iterations as stated in the legend.

5.3 Intermediate conclusion

A prototype has been designed using the algorithms chosen in Chapter 2 to 4. This prototype has been tested to ensure the compliance between the prototype and the functional description of TC. It has been shown that performing more iterations in the prototype result in lower BER, and the compliance between the prototype and the functional description of TC is thereby ensured.

The prototype is therefore the foundation of both the implementation using the soft-core processor and the software implementation aided by a hardware accelerator. However, the generator polynomials and the block size is fixed in the two implementations.

That concludes this part of the report where the concept of TC has been described. The functionalities needed to implement this concept has been analysed and some algorithms fulfilling these functions have been introduced. The algorithms have been summarised into a prototype which is found to be compliant with the overall functional description of TC.

The next part of the report focuses on the mapping from the Algorithmic domain to the Architectural domain. The first chapter describes the mapping of the algorithmic descriptions to a software solution designed to run on a soft-core processor inside a Field-Programmable Gate Array (FPGA). The next chapter analyses this implementation with the focus of determining which parts of the soft-core implementation is most suitable for acceleration in hardware.

Part II

Design and Implementation

6

Software implementation

The prototype, described and tested in Chapter 5, is implemented in a soft-core processor on an FPGA. This implementation is illustrated in Figure 6.1 as the mapping from the algorithmic domain to the architectural domain.

The Turbo encoder is not implemented on the embedded soft-core processor as seen in Figure 6.4. This is done because it is considered to be the part of the system using the least amount of computations, and an implementation hereof is therefore not relevant with regards to the objective of this project.

It is chosen to use the Altera DE2 platform which includes a Cyclone II EP2C35F672C6. The reason for this is that the board is produced for educational purposes, and the documentation is therefore extensive. Furthermore, the use of an Field-Programmable Gate Array (FPGA) eases the design of hardware accelerators. This device makes it possible to describe the needed hardware in a hardware description language and synthesize it into the device rather than building the functionality using discrete components.

Furthermore the Nios II soft-core processor is chosen among the many soft-core processors available, as it is widely used on Altera FPGA boards and well-documented.

The implementation is therefore done so that the encoder, used in the prototype, can be used to generate the data for the embedded system. Also the subsequent processing and representation of data is chosen to be performed on the PC. The communication between the PC and the embedded processor uses RS-232.

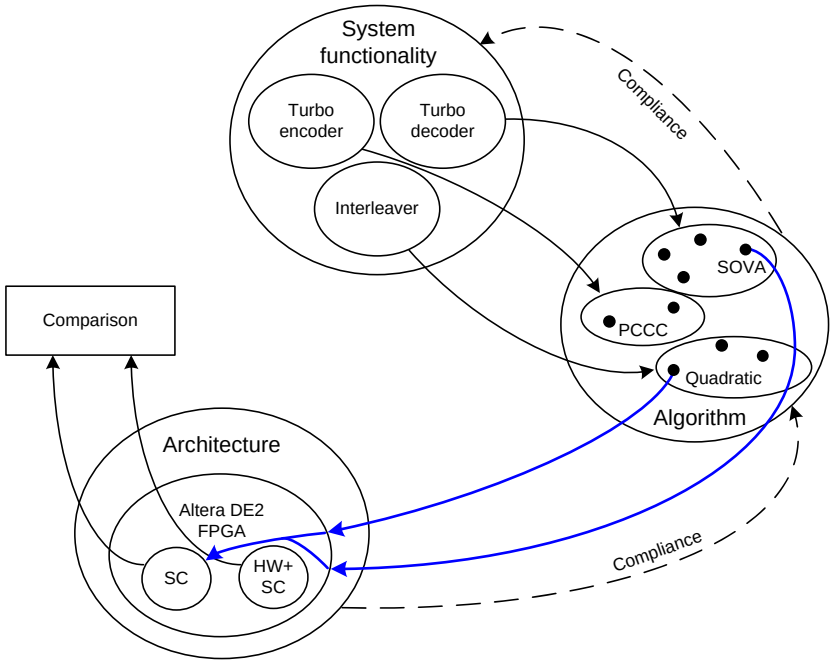


Figure 6.1: Illustration of the current phase of the project with regards to the modified A^3 model. This is the phase for implementation of the Turbo decoder and interleaver/deinterleaver on the soft-core processor.

6.1 C program specifics

In order to design a working implementation of Turbo Coding (TC) on an embedded platform, there are some issues that needs consideration. The issues, and how they are resolved in the implementation, are discussed.

6.1.1 Number representation

First of all, the processor is designed with arithmetic units only capable of integer calculations. This means that one must either use a fixed point representation, or design a floating point implementation using the arithmetic units available. Such an implementation possibly provides a better numerical stability to the TC implementation and certainly eases the design of it. A floating point implementation, however, increases the number of calculations required for rather simple operations, and therefore increases the execution time of the algorithm significantly.

It is therefore chosen to use a fixed point implementation, where all numbers are represented as 32 bit integers with the point placed at 2^{16}. This means that all numbers are multiplied by $2^{16} = 65536$ and rounded to the closest integer lower than the resulting number. The Nios processor, however, is not capable of multiplying or dividing two 32 bit numbers with full precision and range. This is due to the fact that the multiply and divide operations on the processor both input and output 32 bit numbers. A consequence of this, is that one has to choose between range or precision in all multiplication operations. Either one must divide each number by 2^8 (or one of them by 2^{16}) and then multiply the numbers, or multiply the numbers and then divide the result by 2^{16}. The first method throws away 8 bits of precision in each number before multiplying the numbers, whereas the second method can cause up to 32 bits of overflow since $(2^{31} - 1)^2 = 2^{62} - 2^{32} + 1$, which does not fit in a 32 bit register. The software for the embedded

system has therefore been designed to use the best of the two methods or a combination hereof, which is to divide one operand by 2^8 and also divide the result by 2^8.

6.1.2 Nonlinear functions

Another issue to be considered is the fact that the Soft Output Viterbi Algorithm (SOVA) algorithm makes use of both the exponential function and the natural logarithm. These are non-linear functions, and it is therefore not possible to compute these functions precisely for all input values. Two methods are widely used to approximate these functions. The first method is to write a lookup table with function values and use the function value that corresponds to the nearest input. Another method is to substitute the exponential function and the natural logarithm with Taylor expansions of some finite order N.

The problem with the first of these two methods is that a large amount of memory is needed, and that makes it virtually impossible to achieve an acceptable precision. The disadvantage of the second method is that it demands a lot of multiplication operations since both a high power of the input value and a factorial operation is needed. Furthermore, the second method is only valid inside a rather limited range of input values, and outside of this range the polynomial approximation might differ arbitrarily much from the real functions.

It is therefore chosen to use a third method called the Jacobian algorithm [Vucetic and Yuan, 2000, p. 151] which states:

$$\ln\left(e^{\delta_1} + e^{\delta_2}\right) = \max(\delta_1, \delta_2) + \ln\left(1 + e^{-|\delta_2 - \delta_1|}\right) \tag{6.1}$$

This makes it possible to approximate terms on the form $\ln\left(e^{\delta_1} + e^{\delta_2}\right)$ by the largest of the two numbers δ_1 and δ_2 with the error $\ln\left(1 + e^{-|\delta_2 - \delta_1|}\right)$.

By inspecting the last term of the branch cost in the SOVA algorithm (Equation 4.16 or 4.17), it is seen that those terms can be written on the form needed by the Jacobian algorithm:

$$\ln(\mathrm{P}(x_t = 0)) = \ln\left(\frac{1}{1 + e^{L_a(x_t)}}\right) = \ln(1) - \ln\left(e^{\ln(1)} + e^{L_a(x_t)}\right) \tag{6.2}$$

$$\ln(\mathrm{P}(x_t = 1)) = \ln\left(\frac{e^{La_x(t)}}{1 + e^{L_a(x_t)}}\right) = L_a(x_t) - \ln\left(e^{\ln(1)} + e^{L_a(x_t)}\right) \tag{6.3}$$

These terms can therefore be calculated using the Jacobian algorithm, and can be approximated by leaving out the second term. One can also use a second order Taylor expansion, of the second term in the Jacobian algorithm, to get a better approximation of the function. Figure 6.2 shows the Jacobian approximation, both with and without the second order Taylor expansion of the second term (correction term) of the algorithm, together with the function $\ln(1 + e^x)$. Figure 6.3 shows the difference between the real value of the function $\ln(1 + e^x)$ and the approximations with and without the second order correction.

It is seen that using the approximation, with the second order correction function, instead of the

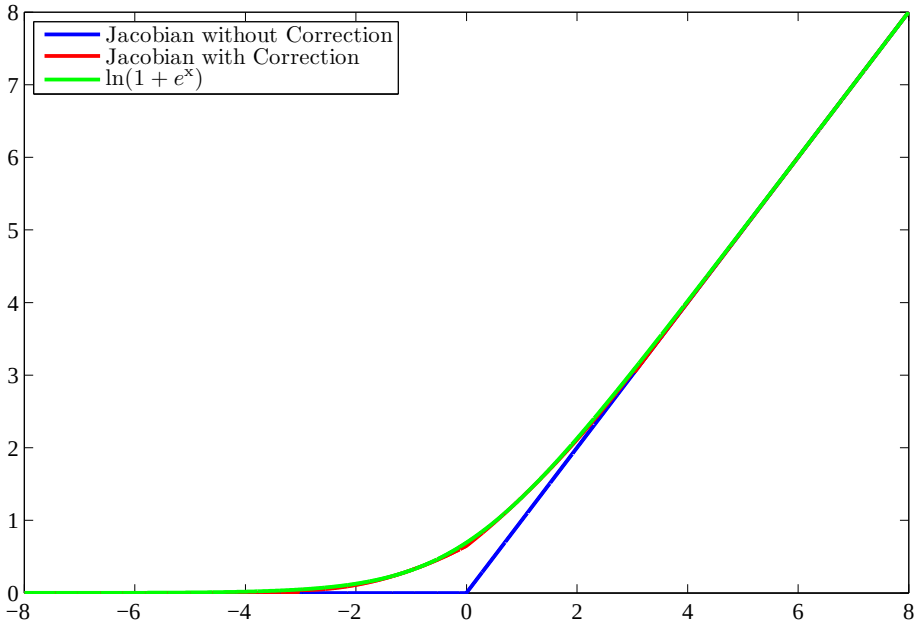


Figure 6.2: Jacobian algorithm used to approximate the function $\ln(1 + e^x)$. Here it is shown both with and without the correction term.

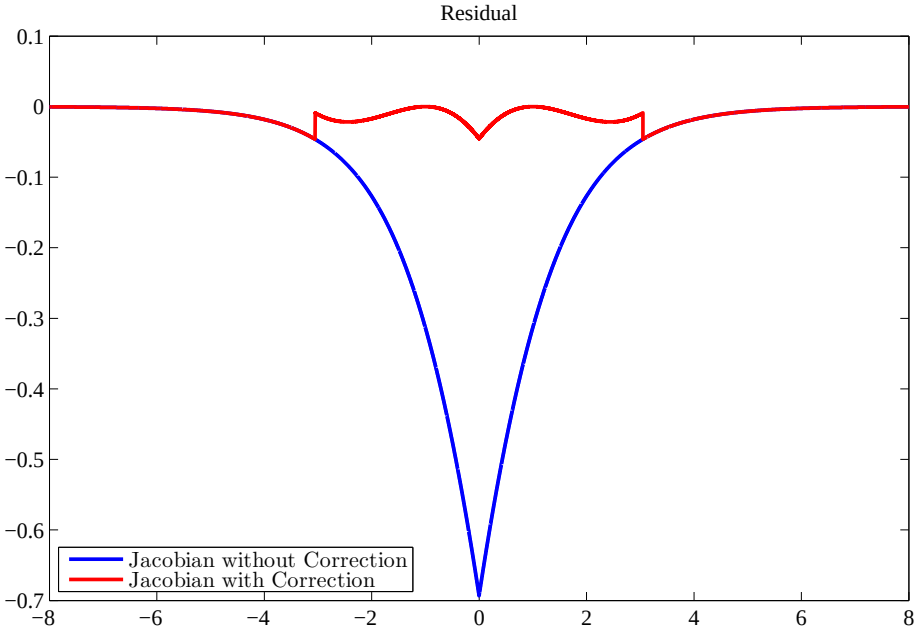


Figure 6.3: A plot of the residual between the function $\ln(1 + e^x)$ and the approximation obtained by use of the Jacobian algorithm. Here it is shown both with and without the correction term.

real function makes a maximum error in the order of 10^{-2}, and it is therefore estimated that this approximation is a reasonable trade-off between computational complexity and precision.

6.1.3 Fixed generator polynomials

When examining the prototype implementation of TC described in Chapter 5, one finds that calculating all the possible outputs of the encoder given the generator polynomials is using a significant amount of computational power. This part of the prototype is only implemented to be able to test the effects of different generator polynomials, and is therefore unnecessary in the final implementations. Leaving out this part, and writing the outputs of the encoder given all states and inputs into memory, significantly improves the execution speed of the system.

6.1.4 The resulting system

The system, resulting from the considerations in the previous sections, is illustrated in Figure 6.4. The software embedded in the Nios II soft-core processor is composed of 5 functions:

`main` The purpose of the `main` function is to receive the data from the PC, place it in the right data structures and call the `decode` function. When the `decode` function is done, `main` transmits the decoded data to the PC. The `main` function measures the execution time of the other functions, and transfers this information to the PC.

`decode` The `decode` function is in charge of the iterative part of the decoding process. This means that the `decode` function calls the functions `interleave`, `deinterleave` and `sova` in the way described in Chapter 4.

`interleave` and `deinterleave` The functions `interleave` and `deinterleave` is in charge of interleaving and deinterleaving either the `La`, `sys` or `LLR` sequence. It has been chosen to implement the quadratic interleaver as described in Section 3.4 and implement the permutation using a lookup table rather than the matrix-vector multiplication.

`sova` The `sova` function decodes as described in Section 4.5.

In order to be able to ensure that the software can execute on the embedded platform, it is necessary to estimate the memory usage of the software, and then to ensure that a sufficient amount of memory is available.

6.1.5 Memory requirements

A long list of intermediate values are needed when designing a system to decode using the TC scheme. The most obvious of these are the extrinsic values $L_{e1}(\underline{\hat{\mathbf{x}}})$ and $L_{e2}(\underline{\hat{\mathbf{x}}})$ and the decoded bit vectors $L_1(\underline{\hat{\mathbf{x}}})$ and $L_2(\underline{\hat{\mathbf{x}}})$. However, also the `sova` and the `main` function utilises a substantial amount of memory.

It is therefore chosen to do a rough estimation of the memory usage in each function of the software design, the result of which can be seen in Table 6.1. In this estimation, only variables containing

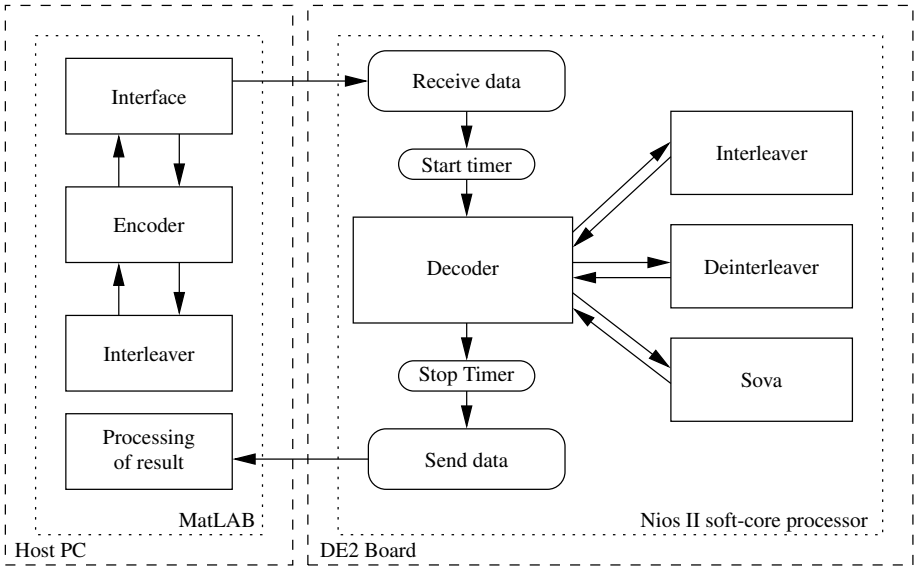


Figure 6.4: Block diagram depicting the way the soft-core software implementation is built in C code.

data used in the calculations, are considered. Variables like loop counters and the like are therefore neglected. Furthermore it should be said that the estimation does not take into account that only some variables are present simultaneously. The estimation is therefore considered as worst case.

Memory requirements		
Module	Usage	Variable name(**data type,array length**)
main.c	36,926 B	iterations(**char,1**), char1(**char,1**), char2(**char,1**), char3(**char,1**), char4(**char,1**), char_sign(**char,1**), passed_time(**int,1**), sys_length(**int,1**), par11_length(**int,1**), par12_length(**int,1**), par21_length(**int,1**), par22_length(**int,1**), sys(**int,2048**), par11(**int,2050**), par12(**int,2050**), par21(**int,2050**), par22(**int,2050**)
math.c	16 B	output(**int,1**), temp(**int,1**), first_term(**int,1**), second_term(**int,1**)
decoder.c	18,432 B	LUT(**int,2048**), sys_int(**int,2048**), dec_data(**char,2048**)
sova.c	81,966 B	outputindex0(**char,1**), outputindex1(**char,1**), outpar1(**int,1**), outpar2(**int,1**), branch0(**int,1**), branch1(**int,1**), logprob(**int,1**), temp_cost(**int,1**), La_temp(**int,1**), temp(**int, 1**), syscost(**int,1**), par1cost(**int,1**), par2cost(**int,1**), mu_best(**int,8192**), mu_best_back(**int,8192**), La(**int,2048**), LLR(**int,2048**)
interleave.c	8,192 B	temp_arr(**int,2048**)
deinterleaver.c	8,192 B	temp_arr(**int,2048**)
Sum:	148,724 B	

Table 6.1: Listing of the memory requirement for each of the C code functions which are implemented. This accounts only for the data storage, not the program by itself, and it does not account for the fact that variables may not require storage simultaneously.

As shown in Table 6.1, around 148,724 B of storage is required at the for data.

As mentioned, this does not take into account the compiled C code in itself. The size of this can be found using the Nios II IDE software which during compiling reports as follows:

”Info: (main.elf) 250 KBytes program size (code + initialized data).”

The amount of memory needed is therefore no more than 250 + 148,724/1024 $\approx$ 395 KiB. It is therefore estimated that 400 KiB of memory will be enough for execution of the software.

The following section describes the choice of platform and design of the soft-core processor implementation.

6.2 Platform specifics

As previously mentioned, it is chosen to use the Altera DE2 Development and Education board. This board features all the peripherals needed and features a Cyclone II EP2C35F672C6 FPGA.

The reason for choosing to test the implementations of TC on an FPGA is that an FPGA is a re-programmable hardware platform. In an implementation, the functionality of a algorithm can be realised by using combinational logic such as AND, OR, XOR etc. gates. In an FPGA, the combinational logic is created by storing a logic '1' or '0' in an SRAM cell which controls the gate of a transistor. By storing multiple values it is possible to synthesize a design of combinational logic elements [Woods et al., 2008, p. 2].

6.2.1 Soft-core and peripherals

A soft-core processor is a collection of components, written in a hardware description language, which can be synthesized for different programmable devices such as FPGAs. A wide range of such processors exists, both commercial and open source and with different architectures. One type, commonly used on Altera platforms, is the Nios II processor which can be customized to fit the purpose at hand by using the SOPC Builder software.

Furthermore, it is possible to define and include all required peripherals for the Nios II processor using SOPC Builder. The components on the DE2 Board can be included in the SOPC Builder design. An overview of some of these and the interconnection structure can be seen in Figure 6.5 [Altera Corp., 2009, p. 19]. The peripherals are interfaced to the processor using the Avalon switch fabric which includes the functionality needed for communicating with the peripherals. The contents of the switch fabric depends on the signals required by the peripherals. For example, the Avalon switch fabric takes care of proper chip-selection and wait state introduction when a program, being executed on the Nios II processor, tries to access a specific address in a memory block.

6.2.2 Requirements

As illustrated in Figure 6.5, it is possible to use many types of peripherals with the Altera DE2 board and the Nios II soft-core processor. This makes the board suitable for many different applications, but many of the peripherals are of no use in this project. It is therefore decided to determine exactly which peripherals are actually required in order to construct the soft-core system in SOPC Builder. Naturally, only peripherals which are avaliable both on the DE2 board

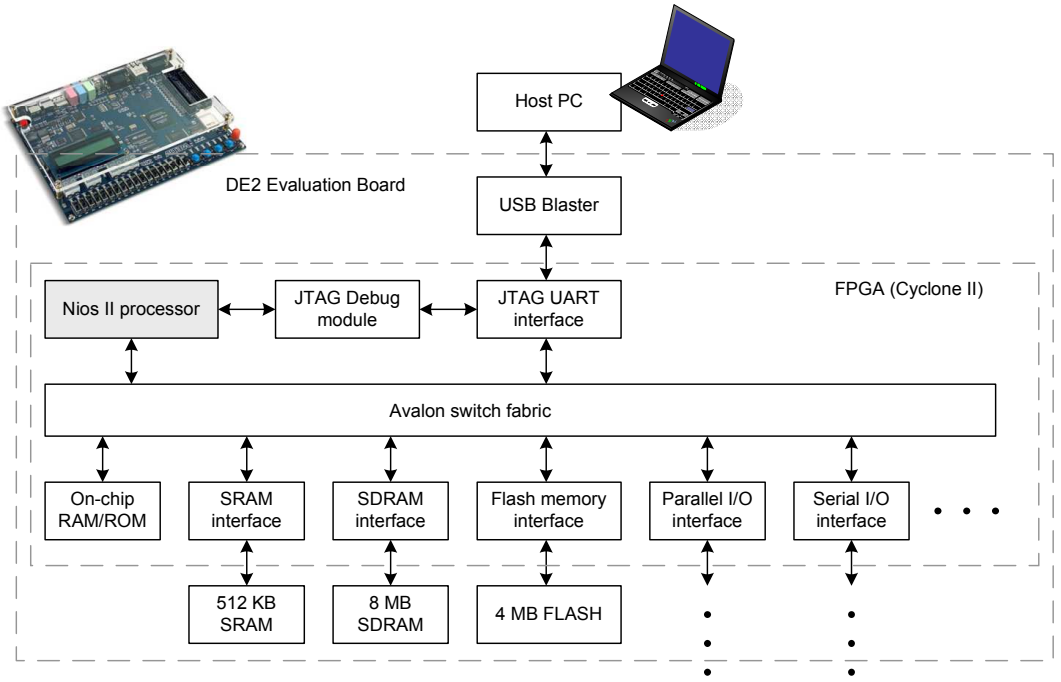


Figure 6.5: Overview of some of the peripherals that can be integrated on the DE2-Board as IP cores using the Altera SOPC Builder software.

and in the SOPC Builder are considered in this discussion.

Memory type

The DE2 board includes several different types of memory. Apart from synthesizing memory elements onto the FPGA fabric, the board offers the possibility to use three different types of external memory: 4 MB Flash, 512 KB SRAM and 8 MB SDRAM. The Flash memory is not considered here, as it is considered to be slower than the other memory types, and it is shown in Section 6.1.5 that 400 KiB of memory is sufficient. The two memory devices considered are therefore SDRAM, a PSC A2V64S30CTP-G7 [PSC, 2005], and SRAM, an ISSI IS61LV25616AL-10TL [Integrated Silicon Solution, 2006].

By analysing the timing specifications in the datasheets for the two memory devices, it has been found that a read cycle in the SDRAM device takes around 41 ns, whereas the read cycle of the SRAM device can be done in 10 ns. Likewise, a write cycle takes 21 ns when using the SDRAM device and 4 ns for the SRAM device. The SDRAM device, however, is capable of communicating using burst mode where it is possible to access an entire column in memory in a single read or write cycle [Hallinan, 2006, p. 493], but this functionality is not supported in the SDRAM controller available in SOPC Builder. It is therefore chosen solely to include the SRAM device in the soft-core processor design.

Communication

As mentioned, the software described in Section 6.1 requires a serial interface for communication with software on the PC. This interface is chosen to comply with the RS-232 specification.

Furthermore, the Nios II IDE uses a JTAG UART interface to program the soft-core processor and transfer the program code. Therefore such a peripheral is also needed in the design of the soft-core system.

Timer

In order to measure the time used for decoding data, it is necessary to include a timer.

From this list of requirements, a host PC-platform connection flow diagram can be established which realises the system setup. This is shown in Figure 6.6.

Nios II processor type

Three different versions of the Nios II processor exist. Which of them to use must be determined from a trade-off between speed (DMIPS/MHz) and the area (Logic Element (LE)) used. The three versions of the Nios II processors are identified by the e, s and f designations. A comparison of these processors is shown in Table 6.2, where some of the most important differences are shown. The table is based on information from [Altera Corp., 2009, p. 115, 119, 128] and the Altera SOPC Builder.

		Processor type		
Feature		Nios II/e	Nios II/s	Nios II/f
Logical elements		600-700	1200-1400	1400-1800
Speed	[DMIPS/MHz]	$\leq$5	$\leq$25	$\leq$51
HW Multiply		No	Yes (5 cycles)	Yes (5 cycles)
HW Divide		No	Optional (4-66 cycles)	Optional (4-66 cycles)
Pipelining	[Stages]	1 (No)	5	6
Shifting		No	Yes (3 cycles)	Yes (1 cycle barrel)
Instruction cache		No	Yes	Yes

Table 6.2: Comparison of the three different Nios II processors implemented on the Cyclone II type.

Even though the Nios II/f variant is using the most area, it is the one selected for this project, since the extra area usage by the Nios II/f out of the total available on the Cyclone II EP2C35F672C6 (approximately 33,000 LEs) is considered not to pose a problem.

Summary

In summary, the processor and peripherals selected for implementation are:

- **Nios II/f:** The fastest of the possible processor variants is chosen.
- **SRAM:** 512 KB available and it has been shown in Section 6.1.5 that only 400 KiB is needed.
- **RS-232 UART:** Communication between MatLAB and the FPGA to send encoded and noisy sequences $\underline{\hat{y}}$ and receive decoded sequences $\underline{\hat{x}}$.

- **JTAG UART:** Interface for loading the Nios II system and IP cores onto the FPGA from the Altera Quartus II software frontend for SOPC Builder and for sending program code using Nios II IDE.
- **Timer:** Timer used for measuring the execution time for a given number of iterations.

This system is illustrated in Figure 6.6.

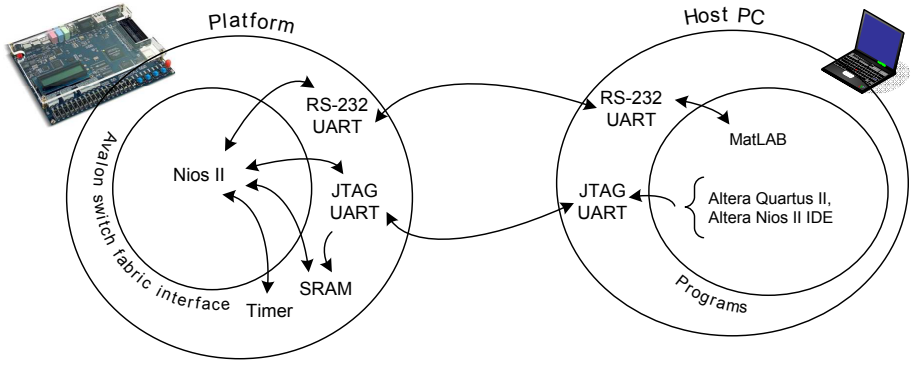


Figure 6.6: Overview of the communication between the host PC and the FPGA platform. Also shown is the internal communication in the platform, where the Nios II is the processor.

6.2.3 Construction and instantiation of system

Launching the Quartus II application and opening the project *turbo_coding_SOPC.qpf* included on the CD, and afterwards launching the SOPC Builder. It can be seen that the peripherals are configured with the following settings, are connected and have the addressing ranges as shown in Figure 6.7. For the SRAM and RS-232 UART IP cores, the DE2 University Program IP Core Library is required.

- **Nios II processor:**
  - Core type: Nios II/f
  - Hardware multiply: Embedded multipliers
  - Hardware divide: Yes
  - Vectors: Reset: up_avalon_sram_classic_0, Exception: up_avalon_sram_classic_0 (default offsets)
- **SRAM:**
  - (Nothing is customizeable)
- **RS-232 UART:**
  - Baud rate (bps): 19200
  - Parity: None
  - Number of: Data bits: 8, Stop bits: 1

- **Timer:**
  - Period: 1 ms
  - Counter Size: 32 bits
  - Presets: Custom
  - Registers: Activate Writable period, Readable snapshot, Start/Stop control bits
  - Output signals: None
- **JTAG UART:**
  - All buffer depths: 64 bytes
  - All IRQ thresholds: 8

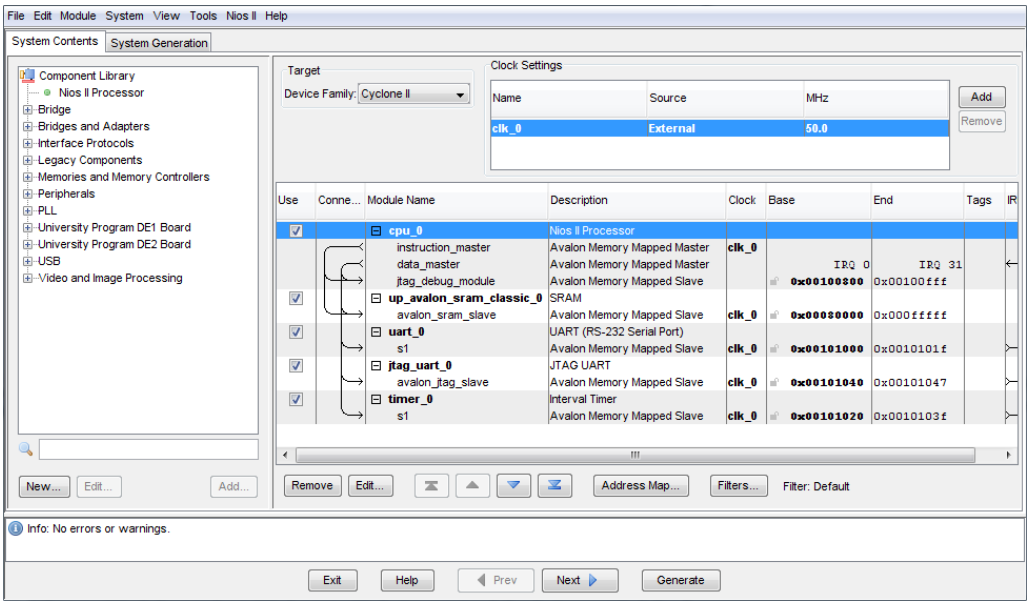


Figure 6.7: The system overview in Altera SOPC Builder, indicating address ranges for the different blocks, IRQs for the communication ports and overall interconnection.

The design requires instantiation code, written in either Verilog or Very high speed integrated circuits Hardware Description Language (VHDL) in Quartus II, in order to synthesize the full system. The instantiation code connects the different port names used by the SOPC builder with the specific pin-names on the FPGA. The VHDL instantiation code used in this project is avaliable on the attached CD.
The system is synthesized and programmed from the Quartus II suite, whereafter the C program is compiled and sent to the Nios II/f using the Altera Nios II IDE program.
After the completion of the synthesis in Quartus II, the summary report states that the total amount of LEs used is 3250 out of 33,216, i.e. roughly 10 % of the available LEs.

The measurement of the execution time for the soft-core implementation is done together with the to-be-presented hardware accelerated implementation, and is carried out in Chapter 8.

7

Applying Hardware Acceleration

Now that the soft-core implementation has been carried out using the previously mentioned methodology, the next step is to examine the implementation to identify the most computationally intensive parts of the algorithm.

This chapter firstly presents the methods and results of determining the computationally intensive parts of the C code, while it is executed on the Nios II/f. Thereafter, the computationally intensive functions are further described using Data Flow Graph (DFG)s. From the DFGs, re-scheduling methods are proposed in order to minimize the execution time for these functions. The proposed re-scheduling of the computationally intensive functions are then implemented in a hardware accelerator written in VHDL.

This is the second and final part of the implementation, where the computationally intensive functions are realised as a separate hardware accelerator, and the implementation therefore forms the combined soft-core with hardware acceleration indicated in Figure 7.1.

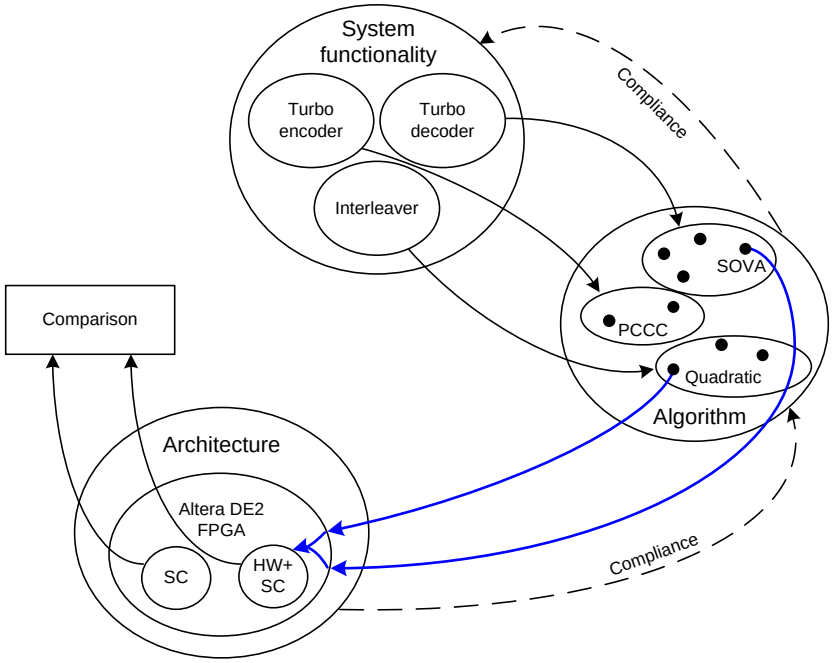


Figure 7.1: Illustration of the current phase of the project with regards to the modified A^3 model. This is the phase for implementation of the computationally intensive parts as hardware accelerators while leaving the other parts of the turbo decoder executing on the soft-core processor.

7.1 Profiling the soft-core implementation

Profiling is a technique applied to program code to gain knowledge about its execution speed. By having separated the program into functions in each C function, the time spent by each function can be determined. The results obtained by profiling can thus help in identifying the most critical functions which could potentially be realised as a hardware accelerator.

While a program can be analysed manually from knowledge of the instructions that the program code is compiled into, it can quickly become an error-prone task to perform, as the program code size increases. The execution time has to be analysed using profiling tools. A profiling tool usually records the number of times certain instructions are executed and the time required for certain functions or blocks of code to execute.

These measurements are analysed in this section and a conclusion is drawn.

Before the analysis of the profiling results for the soft-core processor implementation can begin, the choice and integration of the profiler is discussed.

There are many approaches to profiling and a number of programs using these approaches. The built-in profiler in Nios II IDE is a profiler called *gprof*. In Nios II IDE it is called niosii-elf-gprof [Altera Corp., 2008], and it takes the Nios II soft-core processor architecture into consideration while profiling. This profiler samples the program counter at certain intervals which quickly produces results but it might not give the most precise results.

In order to add profiling cababilities, the software must be linked with the niosii-elf-gprof library. The option for linking the library is available in the the "'System Library Properties"' → "System Library", where the check box "Link with profiling library" must be checked. The timer must be

selected as the ”System clock timer” which is the timer mentioned in Section 6.2.3. Then, when the program code is executed on the Nios II/f soft-core processor, the profiler generates the file ”gmon.out” which contains the profiling data. This file contains the following three sections:

1. Flat profile
2. Call graph
3. Index by function name

The flat profile contains information about the time spent in each function in descending order. The call graph on the other hand shows the information about hierarchy of the child or children function(s) and the number of calls to this/these. The index by the function name is a list of functions which are sorted alphabetically. The data from ”gmon.out” file is read by using the standard text output function from the gprof library. The profiler output is generated by the Turbo decoder functions on the Altera DE2 Development Board using the Nios II/f soft-core processor operating at 50 MHz, as described in Section 6.1.

It is chosen to only present the results for the flat profile, as the other sections only provide information about the hierarchical structure.

Some parts of the results in the flat profile are listed in Table 7.1, where the turbo decoder is performed for one decoding iteration. However, tests show that the increase in time spent in each function for N number of iterations is linear. The detailed report is included on the CD.

Flat Profile results						
% Time	Cumulative seconds	Self seconds	Call	Self s/call	Total s/call	Name
97.66	153.07	153.071	51251	0.00	0.00	altera_avalon_uart_read
0.59	154.00	0.93	2053	0.00	0.00	altera_avalon_uart_write
0.59	154.92	0.92	2	0.46	0.56	sova
0.17	155.17	0.26	51251	0.00	0.00	_srefill
0.16	155.43	0.25	51251	0.00	0.00	read
⋮	⋮	⋮	⋮	⋮	⋮	⋮
0.13	155.85	0.20	98368	0.00	0.00	jacob_log
⋮	⋮	⋮	⋮	⋮	⋮	⋮
0.05	156.19	0.07	1	0.07	156.49	main
⋮	⋮	⋮	⋮	⋮	⋮	⋮
0.01	156.61	0.01	2	0.01	0.01	deinterleave
0.01	156.62	0.01	2	0.00	0.00	interleave
⋮	⋮	⋮	⋮	⋮	⋮	⋮
0.00	156.69	0.01	1	0.01	1.15	decoder
⋮	⋮	⋮	⋮	⋮	⋮	⋮

Table 7.1: Flat profile showing the time spent by selected functions, where some of these belong to the profiler libraries.

In Table 7.1, the % Time is the percentage of execution time of a function with respect to the total program execution time.

The Cumulative seconds is the cumulative sum of the Self seconds. This sum accumulates up to the result in the total execution time.

The self seconds is the time taken by the function alone. In other words, the time has not been accumulated with the other functions. Similarly, calls is the number of times a function is invoked. Self s/call gives information about the time spent by this function per call, whereas Total s/call is the average time used by this function and its child functions per call.

A summary of the flat profile of the functions are given in Table 7.2.

Summary					
Function name	% Time	Self time	Calls	Self s/call	Total s/call
decoder	1	0.01	1	0.01	1.15
Sova	80	0.92	2	0.46	0.56
Interleave	1	0.01	2	0.00	0.00
Deinterleave	1	0.01	2	0.01	0.01
jacob_log	17	0.20	98368	0.00	0.00

Table 7.2: The summary report showing time spent by functions and number of times each of these are called.

Table 7.2 shows the percentage of time spent in the `decoder` C function. The `sova` function, which is the core of the turbo decoder, takes 0.92 s, i.e. 80 % of the time spent in the decoding function described in Chapter 6. The `sova` function is called twice for a single iteration and it takes 0.46 s per call.

Since the `sova` and the `jacob_log` function are using 97 % of the execution time spent on decoding, it is necessary to analyse them to determine if they can possibly be accelerated.

7.2 Identifying part suitable for acceleration

In Section 7.1 it is determined that the `sova` function (including functions called inside) uses $2 \cdot 0.56s = 1.12s$ of the 1.15 s used for the entire decoding. It can therefore be concluded that if the wish is to accelerate the decoding using hardware, the focus should be on this function.

In order to be able to identify the sections of the Soft Output Viterbi Algorithm (SOVA) algorithm fit for hardware acceleration, the overall structure of the software implementation is described and analysed.

7.2.1 Structure of the `sova` function

The content of the SOVA algorithm is described in Section 4.5, but the following section will describe how this is implemented in the C program. The purpose of this description is to have a well-defined basis for the analysis. The source code of the software implementation can be found in Appendix B and the `sova` function can be found on page 106.

In Section 4.5 it is described that both a forward and backward path is needed at each point in the grid in order to be able to calculate the soft output and the extrinsic information for the overall decoder. It is therefore chosen to introduce two large, two-dimensional arrays called `mu_best` and `mu_best_back`. These arrays are used to store the calculated path costs.

The actual computation of the path cost up to and from a given point is performed using two, almost identical loop-structures (one for calculation of the forward paths and one for calculation of the backward paths), where all points in the grid are visited and the branch costs are calculated - one for the case where the last sent value is assumed to be 0 (`branch0`) and one for the opposite case (`branch1`). These branch costs consist of three partial costs and the natural logarithm to the a priori probability (calculated using the Jacobian algorithm as mentioned in Section 6.1.2 and defined in Equation 4.15 on page 39). The first of these partial costs consists of the squared difference between the received systematic bit and a reference value and is thus called `syscost`. Likewise the other two costs are called `par1cost` and `par2cost` because they are calculated from the difference between the received parity-bits and references.

After both the branch costs have been calculated it is possible to calculate the path cost in each point using values in `mu_best` or `mu_best_back` respectively.

When both the forward and backwards paths have been calculated, it is possible to calculate the soft-output and the extrinsic information. This is done using Equation 4.23 and Equation 4.27 on page 43, respectively. For the calculation of these outputs, it is neccesary to go through each point in the grid, calculate both `branch0` and `branch1` and then calculate the full paths through the grid using the values in `mu_best` and `mu_best_back`.

This concludes the description of the structure of the `sova` function, and the analysis can begin.

Flow analysis

It turns out that the `sova` also calls the `jacob_log` function for the Jacobian algorithm. Due to these calls and the amount of memory transactions thus occuring between these functions, the `jacob_log` function is also selected for implementation, even though this function does not account for as much of the execution time as the `sova` function.

This section describes the `sova` and the `jacob_log` functions at a higher detail level, where the method for obtaining this level of granularity is done by the use of combined DFGs and Precedence Graph (PG)s. From these combined DFGs and PGs for the C implementations, methods of modifying these to meet certain execution time constraints are described and completed. Finally, one of these realisation structures is selected for implementation in hardware.

The reason for conducting this analysis is that rewriting code functionality into a hardware accelerator using Very high speed integrated circuits Hardware Description Language (VHDL) is not a straight-forward task, since optimizations can be performed with regards to the way operations are done in a hardware accelerator, compared to a C implementation due to, for example, concurrency. The C code is written to only execute sequentially, i.e. it does not make use of multi-threading [Carver and Tai, 2006, p. 2]. This choice was made since the Nios II/f in this project does not run any operating system supporting multi-threading, nor does it support concurrency [Fort et al., 2006, p. 3], [Labrecque and Steffan, 2007, p. 2].

Design space

Optimizations, which are done in this project with regards to trade-offs between cost metrics, are done between execution time, T, and number of arithmetic operations ((+, –, $\times$, /, compare, shift), hereafter operations) taking up FPGA area, A. These trade-offs can be illustrated by points in a two-dimensional space, where each point illustrates a design solution. An example of this is

illustrated in Figure 7.2, containing 7 different solutions.

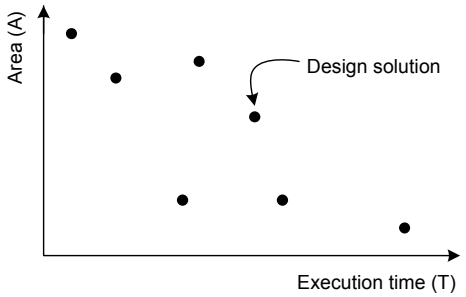


Figure 7.2: Illustration of solutions in the design space, where execution time and area are the cost metrics. The amount and positions of the solutions are merely examples, and does not represent ones specifically for this project.

While only these two cost metrics are considered, theoretically, more cost metrics can be added. Each added cost increases the dimensionality of the design space by one. Due to the increase in dimensionality, more complex comparisions of the possible solutions must be conducted. This can make the evaluation of the design space a long, exhaustive search for the solution with the best trade-offs. Methods exist to aid in the comparisons of the possible solutions, some of which are described in e.g. [Gries, 2004], which also covers a brief description of the available software tools to help the design space exploration process.

In order to choose the solution which is most suitable for implementation, it must be emphasised that the objective of the project is to evaluate the reduction in execution time by introduction of a hardware accelerator. In other words, the area is not constrained, although it is limited by the number of Logic Element (LE)s left on the Cyclone II FPGA, after the soft-core implementation has been done as described in Section 6.2.3.

To initiate the exploration process, the `sova` and `jacob_log` functions are analysed using combined PGs and DFGs.

7.2.2 Data flow and precedence for the C implementation

As noted in the previous section, a C program is normally executed sequentially only if it is single-threaded, which is the case for the specific implementation of SOVA and the logarithm function. In a sequenctially executed program, only one path exist, which at the same time constitutes the critical path.

Even though values may be computed at an early stage in a C function and saved to a variable, the variable may not be requested again until after a number of operations.

The `sova` function consists of three for-loops each executing once per point in the grid i.e. $3 \cdot 2048 \cdot 4 = 24576$ times. Two of the for-loops are almost identical, and computes the best forward and backwards path costs of the trellis, respectively. In comparison, the last for-loop requires a few different operations to compute the outputs based on the calculation of the best forward and backwards path costs. Due to this similarity, the computation of the best forward and backwards path costs are treated together from now on.

The further analysis of the `sova` and `jacob_log` functions are done separately in order to more easily keep a sense of perspective of the discussion.

Best forward and backwards path cost computations

The operations in the computation of the best forward and backwards path cost are illustrated in the combined DFG and PG in Figure 7.3.

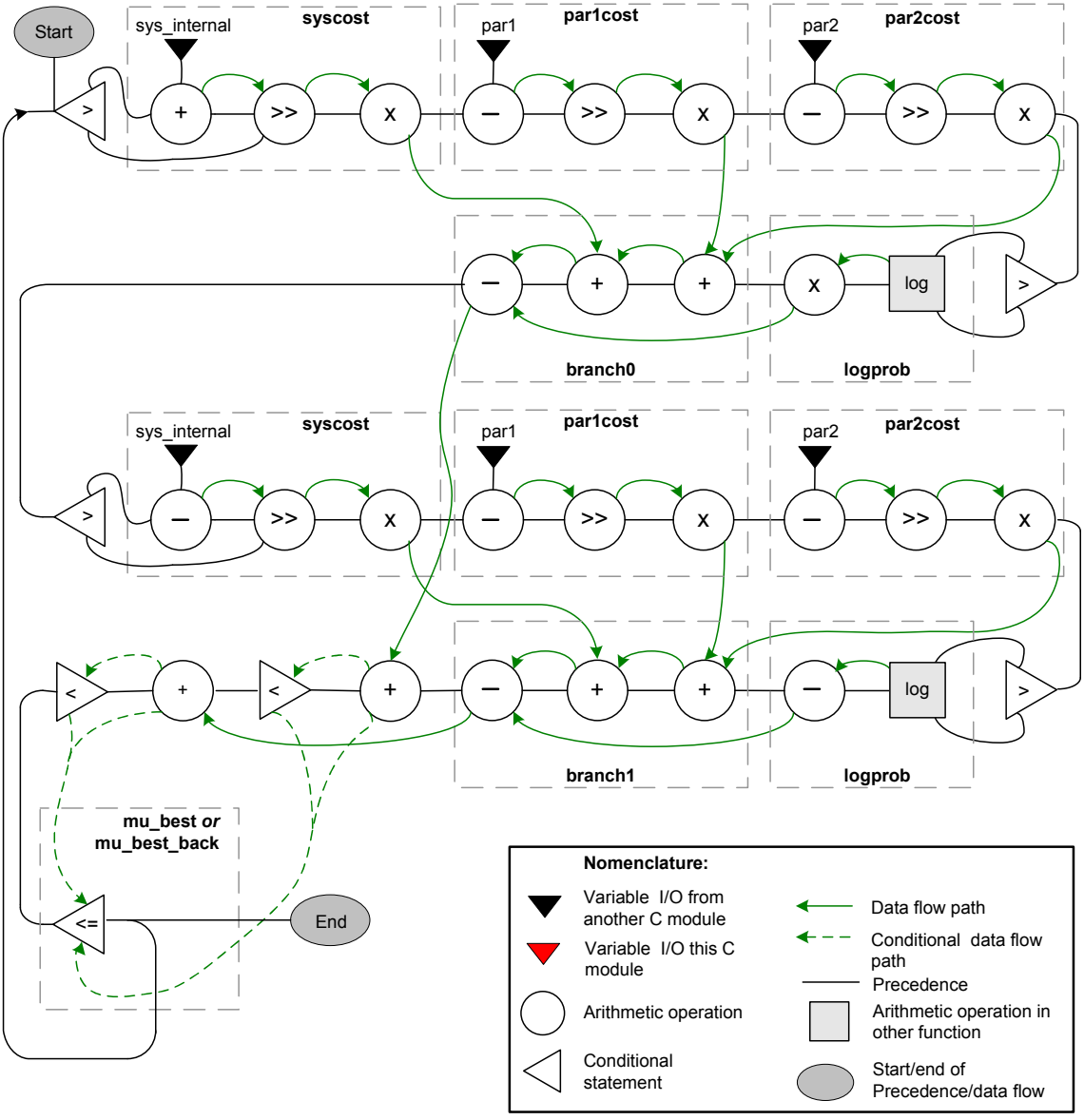


Figure 7.3: The PG and DFG of the backwards and forward evaluation of the path cost. This process runs 8192 times for computation of all elements in `mu_best` and `mu_best_back`.

In the `sova` function, a comparison statement could also be displayed by a circle, like the other arithmetic operations. However, in order to make the PG more readable when comparing it with the C code, it was chosen to select a different symbol for these.
The two nodes denoted by ”log” call the `jacob_log` function and corresponds to more than a single operation.

Computation of output

The combined DFG and PG for the output computation of the `sova` function is seen in Figure 7.4.

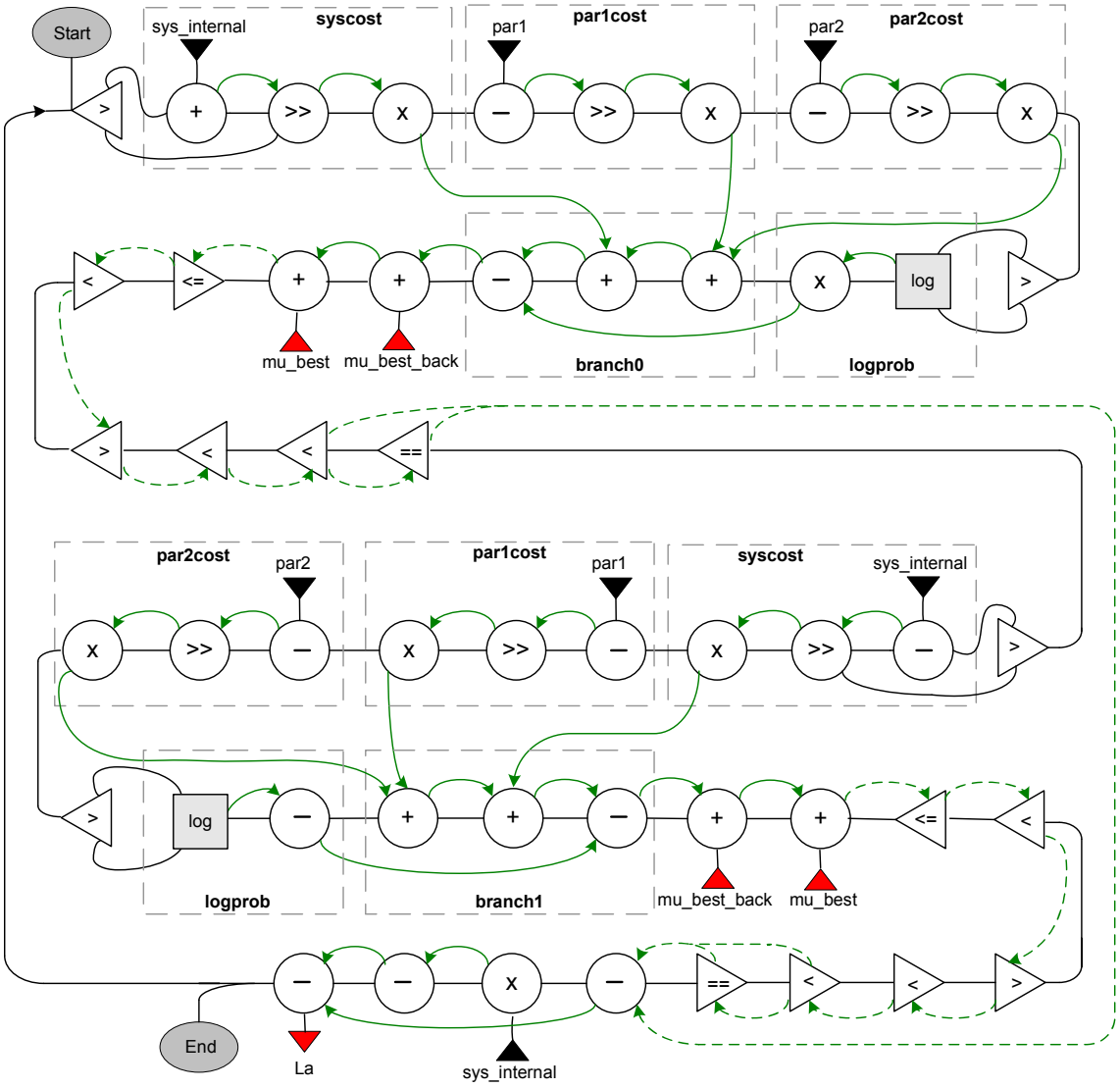


Figure 7.4: The PG and DFG of the output computation from the branch metrics. This algorithm runs 8192 times and will output the extrinsic values (La) and the soft-output (LLR) - one per four possible state transitions.

By examining Figure 7.4, it appears that, as mentioned before, many of the operations of Figure 7.3 are kept, with just a few new operations and comparisons and a few removed. The `branch0` and `branch1` metrics are recalculated, due to the large amount of memory required if it should have been saved.

Computation of `logprob`

The `jacob_log` function, which is called by `sova` function, is shown in Figure 7.5 as a combined DFG and PG. After a number of comparisons the PG can follow one of two paths depending on the value of the input to the function. Unlike the previous two PGs, this function has two paths through the PG.

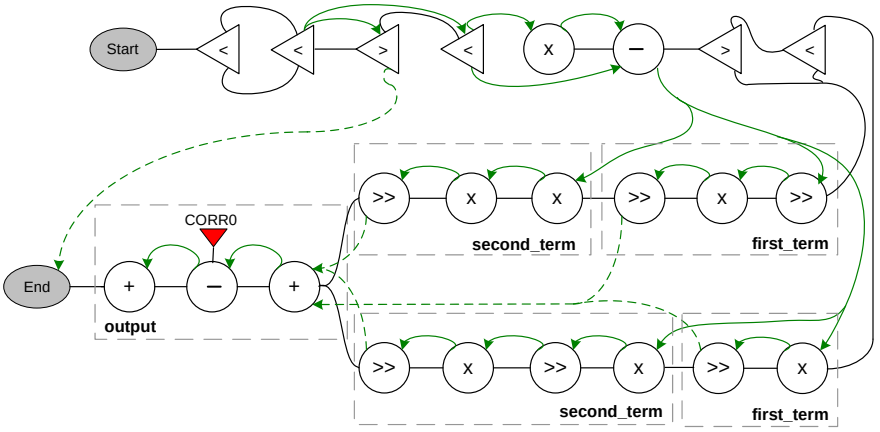


Figure 7.5: The PG and DFG of the `jacob_log` function which is called from the `sova` function in order to calculate the branch costs, `branch0` and `branch1`. More specifically, the logarithm function is called in order to calculate `logprob`.

Design space extreme 1/2: Sequential execution

If the branch costs and logarithm computations were done in a hardware implemententation using the sequential structures of the presented DFGs and PGs, the solution can be considered as one that takes a relatively long amount of time to complete, compared to the area required to realise this. The area consumption is small, since due to the sequentiality, no simultaneous operations are required and all functional units can be reused.

The objective is now to decrease the execution time, by indentifying the inherent parallelism, which is present in the DFGs, presented above, for the `sova` and `jacob_log` functions.

7.2.3 Identifying inherent parallelism

The inherent parallelism in the presented DFGs of Figure 7.3, 7.4 and 7.5 is seen from observing the patterns in the data flow.Some observations can be made from the three previous figures, which all regard parallelism:

Number of computations:

1. There are $2 \cdot 2,048 \cdot 4 = 16,384$ executions of the algorithm in Figure 7.3 to calculate `mu_best` and `mu_best_back`.
2. There are $2,048 \cdot 4 = 8,192$ executions of the algorithm presented in Figure 7.4.
3. Due to the fact that the logarithm function is called twice for each branch computation (forward path, backwards path and output evaluations), it is executed $2 \cdot (16,384 + 8,192) = 49,152$ times.

Dependency:

4. The computation of `branch0` and `branch1` is not possible before the computations of `par1cost`, `par2cost` and `logprob` have completed.
5. The algorithm of Figure 7.4 requires knowledge of `mu_best` and `mu_best_back` to calculate a temporary cost, right after the calculation of `branch0` and `branch1` (still in Figure 7.4).

Independency:

6. The order of calculation of `branch0` and `branch1` is irrelevant.
7. The order of calculation of `syscost`, `par1cost` and `par2cost` is irrelevant.
8. The order of calculation of `first_term` and `second_term` is irrelevant.

Rescheduling best forward and backwards path cost computations

Keeping these considerations in mind, and by noting that independent parts of the algorithm can be executed in parallel, the rescheduling of the algorithm for the computation of forward and backwards path costs in the `sova` function, can be performed like shown in Figure 7.6.

Note the colored circles in Figure 7.6, which serve as identifiers for the respective dashed areas of the algorithm that the circles are inside. Representations using these symbols as substitutes for the areas will be used from now on. Figure 7.6 is constructed from properties 1, 4, 6 and 7.

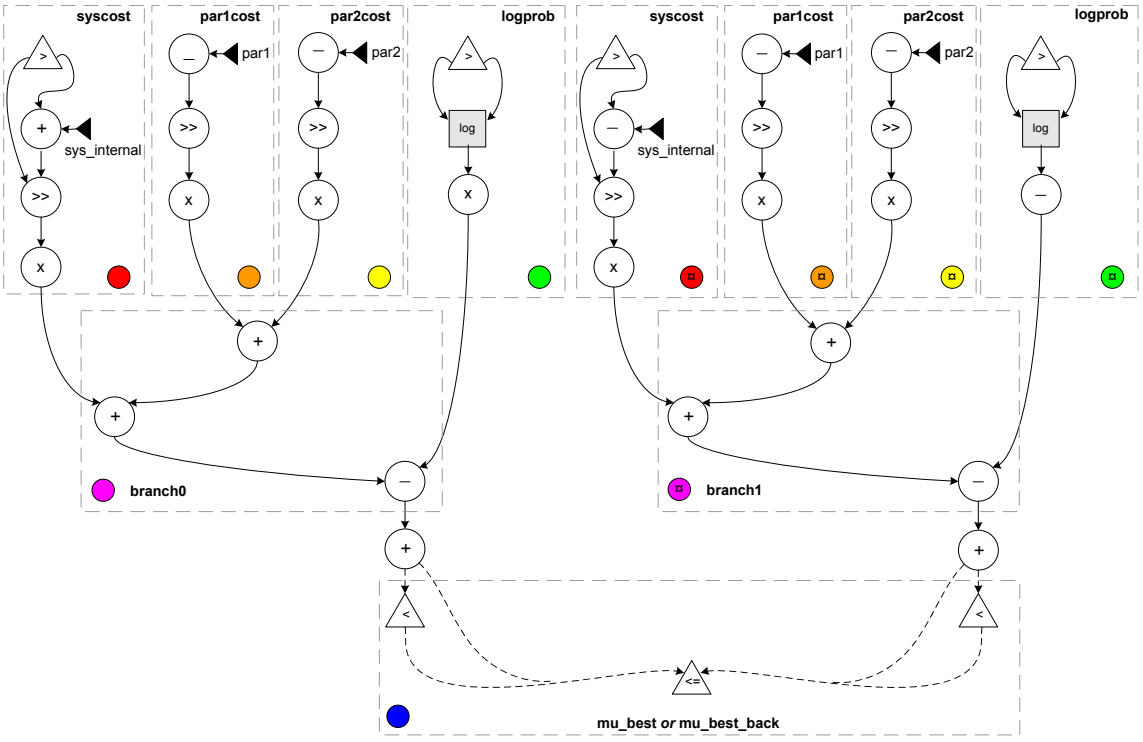


Figure 7.6: Structure of re-scheduling the sequential code to make the inherent parallelism more visible.

Rescheduling output cost computation

Figure 7.7 shows the second part, where the branch metrics are re-computed and used with `mu_best` and `mu_best_back` from Figure 7.6 to calculate the extrinsic value, La, and the soft-output, LLR. The figure is based on properties 4, 5, 6 and 7 on page 72.

Note the colored triangles, which serve as identifiers for the respective dashed areas of the algorithm that the triangles are inside.

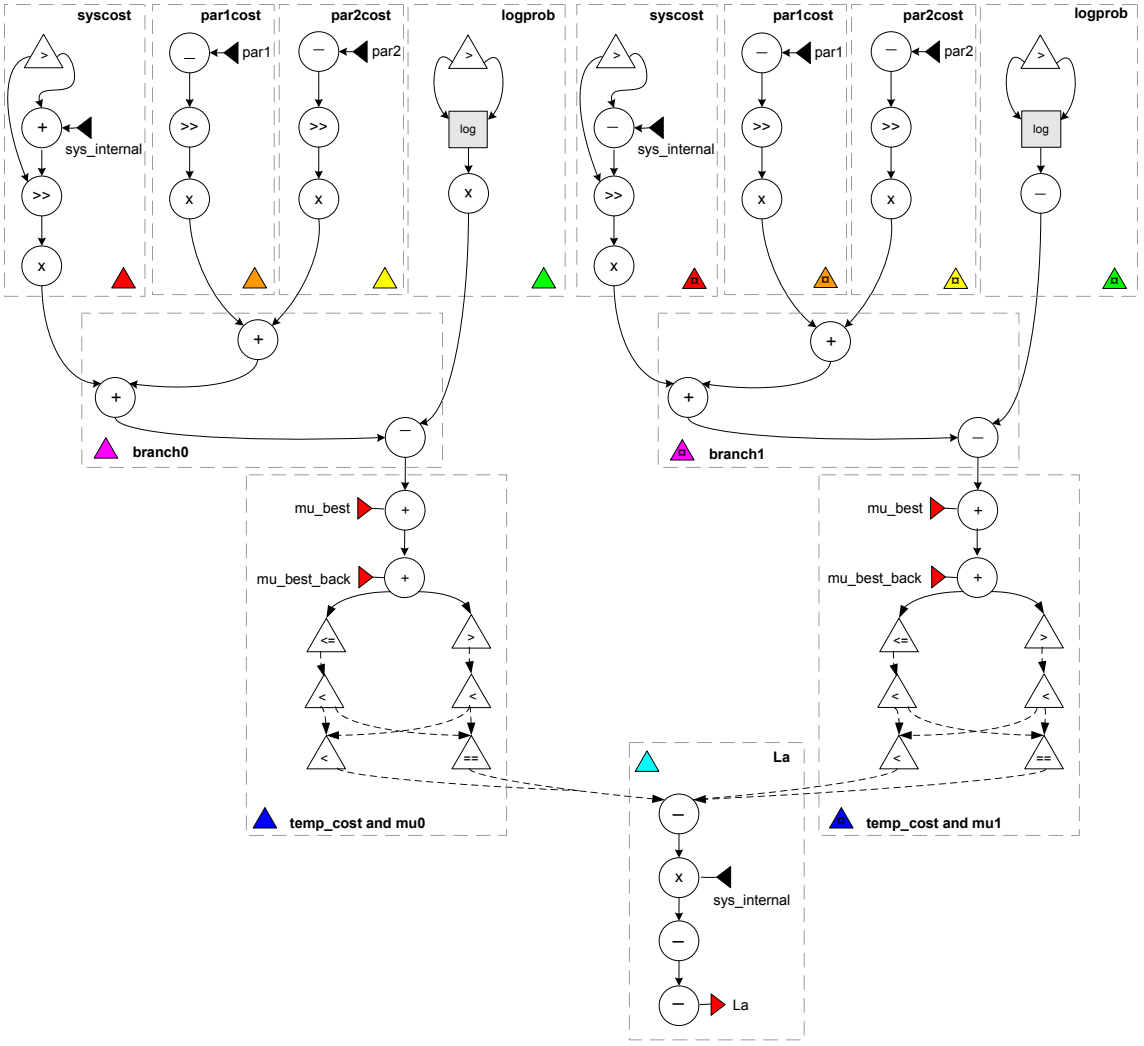


Figure 7.7: Structure of re-scheduling the sequential code to make the inherent parallelism visible. This is for the second part of the SOVA algorithm, where the extrinsic values, La, and the soft-outputs, LLR, are calculated

Rescheduling of the `jacob_log` function

The last rescheduled function, depicted in Figure 7.8, is constructed based on property 3 and 8. In this case it is seen that many of the logical ”and” statements result in no possibilities for parallelization in the beginning of the algorithm. However, parallelism appears which results in a reduced critical path compared to Figure 7.5.

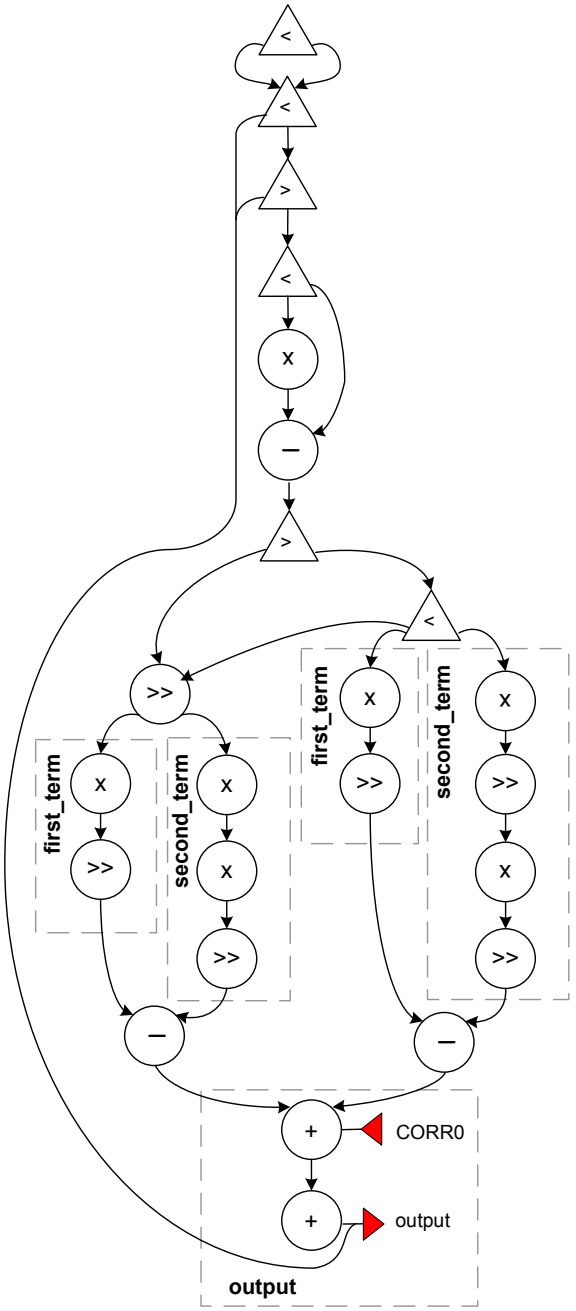


Figure 7.8: Structure of re-scheduling the sequential code for the `jacob_log` function to make the inherent parallelism more visible.

Design space extreme 2/2: Full parallelization

The inherent parallelism of the three parts of the SOVA algorithm has now been studied, and another extreme solution in the design space can be consired: full parallelization. This entails unfolding the for-loops and making an implementation that enables all independent operations to execute simultaneously. However, this is easily discarded as a feasible solution as it would require more than 200,000 multipliers and an equally large number of adders, subtracters etc. Full parallelization is therefore not a realistic solution and, as expected, a solution must be found in between the two mentioned design space extremes.

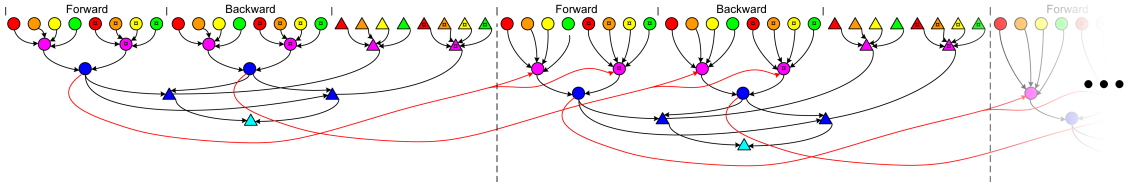


Figure 7.9: Structure of the loop-unfolding model for calculation of branch costs, best paths and outputs. Due to dependices from earlier iterations, the time for computing the extrinsic values La increases by one block of `branch0` and `branch1` for each parallel entity consisting of the computation of forward, backwards path cost and outputs.

7.2.4 Selected implementation

As described, the three loops for computing the best forward path, best backwards path and outputs have certain similarities as can be seen in Figure 7.6 and 7.7. These similarities are taken into account, and it is selected to implement a hardware accelerator with the functionality shown in Figure 7.10.

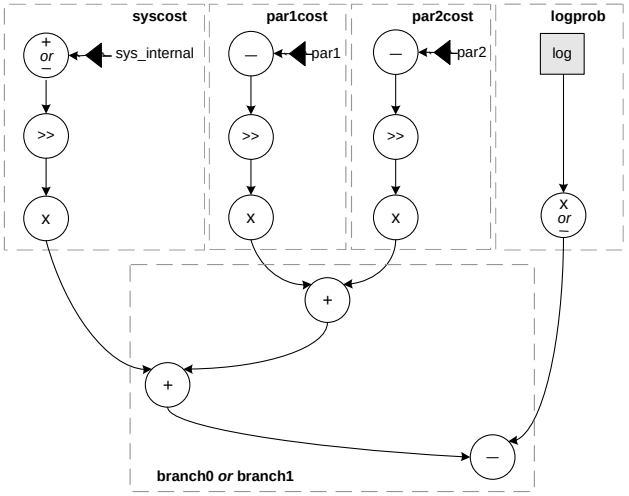


Figure 7.10: The selected parts of the algorithms for the hardware implementation.

The accelerator will be able to compute the branch costs for all three loops, and it can therefore function as a generic block.

This is one solution trading off area and execution time, but it can easily be scaled as several accelerators could be implemented and offer branch cost computation.

When deciding on the implementation, execution time of the selected blocks is the primary deciding factor, but the area constraint of the chosen FPGA is also taken into account.

7.3 Constructing the hardware accelerator

In the previous section, an analysis of the SOVA algorithm has shown that it consists of three loops. The first loop computes the forward path costs while the second loop computes the backwards path costs. The last loop computes the full path costs of all possible paths and the extrinsic and estimated output. It has furthermore been shown, that each of these loops contain identical elements, namely the calculation of a branch cost. It is therefore chosen to implement the calculation of such a branch cost in hardware, and use that to accelerate the software solution.

Figure 7.10 illustrates the computation of such a branch cost with all the parallelism possible. It is therefore implementated using VHDL. Such an implementation can make use of different approaches as described next.

7.3.1 Approach

A hardware platform such as an Field-Programmable Gate Array (FPGA) is, by nature, able to operate concurrently. It consists of gates that compute a new output when the input changes. By implementing several gates side by side, it is possible to achieve a very high degree of parallelism. Furthermore, a hardware implementation need not be constrained by the clock frequency. Each of the gates computes the new value as soon as possible, which can be much faster than the clock frequency.

This leads to two different approaches for implementing the branch cost calculation in hardware. One implementation approach uses the two properties just described (concurrency and independency of the clock), to calculate the result as fast as possible. This approach, however, consumes a large amount of area as there is no reuse of gates.

Another approach is to combine groups of gates to form functional units such as multipliers, Arithmetic Logical Unit (ALU)s and the like, and then use a Finite State Machine (FSM) to control the functional units and make them perform the correct algorithm. This approach reuses the gates but the calculations needs to be coupled to the clock signals which will increase the execution time.

As previously mentioned, area usage is not an issue in this project, as long as the implementations fit on the available area in the FPGA. It is therefore chosen to use the first of the two approaches, since is offers the lowest execution time.

In order to use sequential logic such as in-statements and the like, it is necessary to use the VHDL concept of processes [Rushton, 1998, p. 143]. All statements in a process are synthesised in gates connected in a sequential fashion. However, the output is recomputed each time the input to a process is changed. The conditional statements, such as if-statements and the like, controls how the gates in the process are interconnected, and this type of statements are only recomputed each time a input in the sensitivity list of the process are changed.

It is therefore chosen to partition the system into VHDL processes. These processes are illustrated

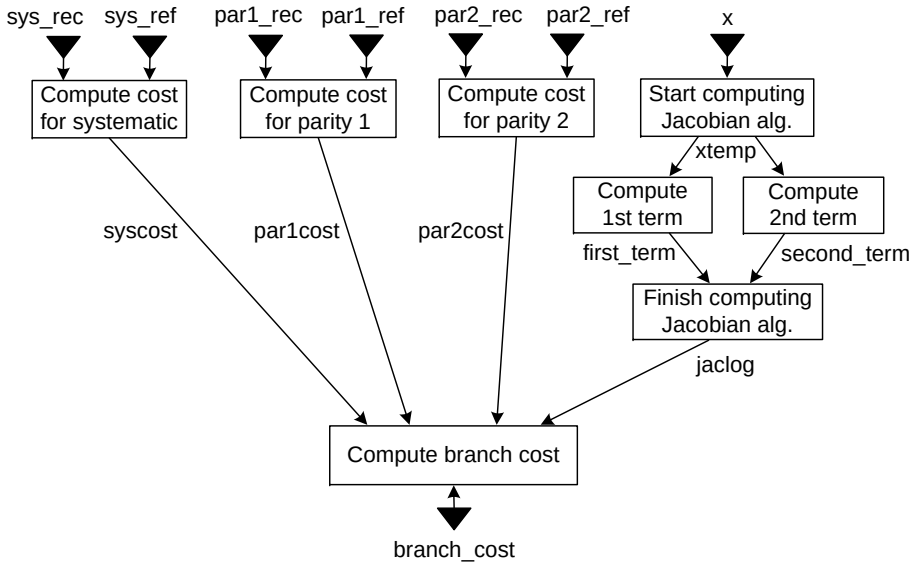


Figure 7.11: Overview of the functional processes in the hardware accelerator. Square boxes represent processes and arrows represent signals.

in Figure 7.11.

All the processes illustrated in Figure 7.11, have all input values in their sensitivity list such that a new output is calculated each time the input changes, and such that this output is correct with regard to the conditional statements in the process.

Figure 7.11 only illustrates the processes needed for implementation of the computation of the branch cost. Another process is, however, necessary to make the hardware accelerator able to communicate with the Nios II processor using the Avalon Switch Fabric. The content of this process is described first.

7.3.2 The communication process

The communication interface is a process in the same architecture as the processes needed for calculation of the branch cost. This process has the sole purpose of keeping track of reading and writing to and from the Avalon switch fabric and thus feeding the other processes with data and returning branch cost after the calculation is done. The hardware accelerator is included into the SOPC Builder just as any other components. This means that the component is memory mapped and is given an address range to which it must respond. Figure 7.12 shows how the component is connected to the processor, and which signals must be considered when designing the interface.

As illustrated in Figure 7.12, the communication process should be sensitive to the clock. This means that each time the clock goes high, the process should test if either the read or the write signal is high. If the write signal is high, it means that the Nios II soft-core processor wishes to send data to the hardware accelerator, and the communication process should therefore receive the data from the data bus and place is on the right signals. Which of the six needed variables, is sent, is indicated by the address offset from the base address and can be read on the 6 least significant address signals.

The original sova.c code, for the soft-core implementation, must be rewritten to make use of calls

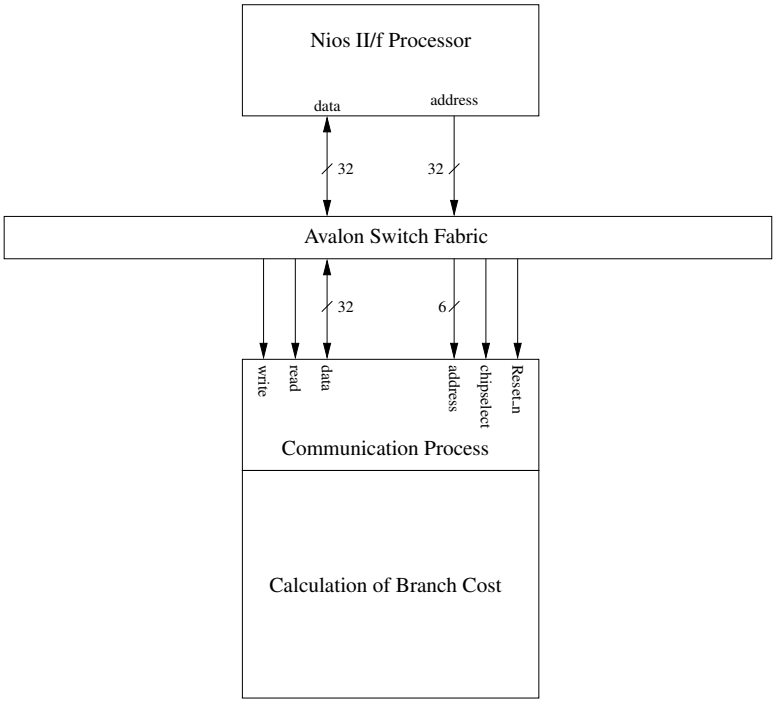


Figure 7.12: Overview of the communication signals and communication process in the hardware accelerator for interfacing to the Avalon bus.

to hardware. These calls are possible, through macros defined in the header file io.h. Since the data bus is 32 bits wide, the most convenient macro is one that reads or writes 32 bits by a single call. In order to write to the hardware accelerator, the following code is required:

```
IOWR_32DIRECT(BCC_BASE,increment,data);
```

Where:
`BCC_BASE` is the base address for the hardware accelerator as reported by SOPC Builder.
`Increment` is the address at which input data should be written.
`Data` is the data to write to the BCC.

Similarly to read the result from the BCC, the following code is used

```
IORD_32DIRECT(BCC_BASE,increment);
```

By using these two macros, it is possible to rewrite the SOVA algorithm in the sova.c as displayed on page 106, into a new SOVA implementation, hw_sova.c which has the source code as shown from page 110 to page 111.

7.3.3 Implementation

In general, the hardware accelerator has been implemented to achieve similar functionality as the original part of sova.c. In some cases, though, it is possible to achieve higher range and better precision in the hardware implementation. This causes the hardware accelerator to use more operations than the soft-core implementation, but the overall implementation is still expected to be more efficient than the software implementation.

The VHDL source code can be seen in Appendix C.

VHDL example

Below is a short example of VHDL code where two 32 bit signals are to be multiplied and returned in a third 32 bit signal. The point should be at 2^{16} so there are 16 bits for range and 16 bits for precision.

The library used in the example is the IEEE, which offers the `std_logic` type and operations using this type. The type `std_logic_vector` is an array or concatenation of multiple `std_logic`. The two signals, to be squared, are `a` and `b`, which are input ports of type `std_logic_vector` and of size 32 bit. The results is the output port `c` of same type and size as `a` and `b`.

A single constant named `ones` is defined in the architecture for inserting a desired number of ones in a signal or variable. The multiplication of a and b is carried out at line 18-22, where the resulting sign is concatenated with the multiplication of the magnitudes. The multiplication of two 31 bit magnitudes could potentially overflow if returned in 31 bits, so `temp` is actually 61 bits wide. This is neccesary to fit the result in the 32 bit output signal with the point at 2^{16}, so the result needs to be shifted down 16 bits and an overflow check should be performed.

In line 23-27 it is checked whether the result, after shifting, is larger than 2^{31}-1, and is set to 2^{31}-1 if this is the case. The resulting magnitude is in position 16 to 46 of `temp`, and the sign is in position 61. The result is then returned to the signal `c`.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.std_logic_unsigned.all;
4
5  entity example is
6    port(
7      signal a,b : in std_logic_vector(31 downto 0);
8      signal c : out std_logic_vector(31 downto 0);
9    );
10 end example;
11
12 architecture bevaiour of example is
13 constant ones : std_logic_vector(31 downto 0) := std_logic_vector'("
     11111111111111111111111111111111");
14 begin
15   process (a, b)
16     variable temp : std_logic_vector(61 downto 0);
17     begin
18       if a(31)=b(31) then
19         temp := '0' & a(30 downto 0)*b(30 downto 0);
20       else
21         temp := '1' & a(30 downto 0)*b(30 downto 0);
22       end if;
23       if temp(60 downto 47) > 0 then
24         c <= temp(61) & ones(46 downto 16);
25       else
26         c <= temp(61) & temp(46 downto 16);
27       end if;
28   end process;
29 end
```

The above example illustrates how sequential statements are written in a process, and how the in- and outputs are passed. In the example, and throughout the actual implementation, attention is directed towards avoiding overflow, and if it is unavoidable the results are rounded to the maximum

value of the given range (most frequently 2^{31}-1).

7.3.4 Testing component

The VHDL component has been tested in Quartus II by performing a functional simulation with the input signals defined in a Vector Waveform File. This is done to make sure that the implementation functions correctly. The processes for calculating the Jacobian algorithm was tested separately with the inputs 0, $\pm$0.1, $\pm$1, $\pm$2, $\pm$3, $\pm$4 and $\pm$5, which verified all possible paths through the processes of the Jacobian algorithm. The results of the computations are similar to the ones in the C implementation of the Jacobian algorithm which can be seen in Figure 6.2.

The entire branch cost computation was tested with received values of 0, $\pm$0.5, $\pm$1 and $\pm$1.5 compared to the reference values -1 and 1, which resulted in the expected costs. The VHDL component has the required functional capabilities, so it can replace the branch cost computation previously implemented in the C program.

7.3.5 Component attributes

A timing simulation was also carried out in Quartus II, with a Vector Waveform File as in the functional simulation. The timing simulation showed that in most cases the signal containing the branch cost is ready for reading after about 35 ns which is 1.75 clock cycles. To ensure the reliability of the output, it is chosen to wait for 3 clock cycles until passing the values to the soft-core processor.

As stated in Section 6.2.3 the pure soft-core implementation took up 3,250 LEs in the FPGA, which corresponds to roughly 10 % of the total number of LEs. When adding the hardware accelerator, the area consumption increased to 5,909 LEs, which is 16 % of the total area consumption and thus an increase by 2,659 LEs corresponding to 81.82 %.

8 Comparison

In Chapter 5, the prototype is described and tested, and in Chapter 6 it is rewritten to a software solution for the Nios II soft-core processor. The processor is embedded in an Altera Cyclone II FPGA on the DE2 Development and Education board. It is then described, how this software solution can be aided by hardware acceleration, and how such an accelerator is designed. The analysis and description of this is described in Chapter 7.

Both of the necessary mappings from the algorithmic domain to the architectural domain are therefore now complete. This chapter will discuss the comparison of the two solutions. The purpose of this chapter is to specify the test and present the results from the soft-core and combined soft-core and hardware implementations to be able to compare the performance with regards to execution time, while still making sure that the functionality of the turbo coding principle is preserved. After the test is specified, the results are presented.

8.1 Test specification

The test is specified in order to provide the required data to be able to compare the two different implementations of Turbo Coding (TC), with regards to execution time, while also verifying that the turbo coding principle is preserved. This test is specified as follows:

1. Generate a random bit sequence of length 2048 using MatLAB.
2. Add white Gaussian noise as described in Chapter 5 and ensure an SNR level of 2 dB.
3. Send the bit sequence to both implementations and make them decode with the number of iterations ranging from 1 to 20.
4. For both implementations the time used for decoding is measured. The time is measured in exactly the same way in the two implementations to remove bias. The time, along with amount of iterations, should be noted.

An execution of the test above results in the time it takes for both of the implementations to compute 1-20 iterations of the iterative decoding loop. These results are plotted, and it is possible to identify if the hardware acceleration of the Soft Output Viterbi Algorithm (SOVA) and the Jacobian algorithm lower the execution time in comparison with the soft-core implementation.

A second test is performed in order to evaluate if the implementations in hardware complies with the functional description of TC. It is chosen to perform the test with same parameters as the test described in Chapter 5, since the purpose of this test is the same, and the results should be comparable. All the different implementations (including the prototype) are fed with a random sequence of data with noise added to ensure Signal-to-Noise Ratio (SNR) levels ranging from -8 to 2 dB with a step of 1 dB. Unlike the test in Chapter 5, this test is only repeated 20 times and for the number of iterations ranging from 1 to 3, due to the increased duration of the tests on the embedded implementations. The Bit Error Rate (BER) of the decoded data is then computed and the BER is plotted in graphs similar to Figure 5.3 on page 48.

8.2 Results

The tests are performed using the setup illustrated in Figure 6.6 on page 60 with and without the hardware acceleration. The results of the first test, testing the execution time between the two implementations, are shown in Figure 8.1 and the data points are shown in Table 8.1. The prototype is not included in this test, since the execution speed depends on the host PC on which it is executed.

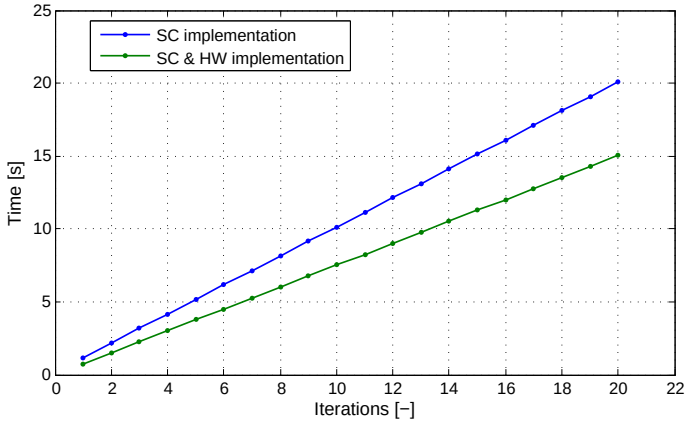


Figure 8.1: Results for the execution time between the soft-core and the soft-core with hardware acceleration implementations.

Test 1: Computation time [s]		
Iterations [-]	Soft-core	Soft-core & Hardware
1	1.1700	0.7670
2	2.1820	1.5190
3	3.1790	2.2700
4	4.1740	3.0200
5	5.1680	3.7710
6	6.1630	4.5220
7	7.1560	5.2730
8	8.1500	6.0240
9	9.1450	6.7740
10	10.1390	7.5250
11	11.1330	8.2760
12	12.1270	9.0270
13	13.1210	9.7780
14	14.1150	10.5280
15	15.1100	11.2790
16	16.1040	12.0300
17	17.0980	12.7810
18	18.0920	13.5320
19	19.0860	14.2820
20	20.0800	15.0330

Table 8.1: Results of the first test, where execution time is presented with the number of decoding iterations.

The result of the second test, where the BER is tested, is seen in Figure 8.2. Data points for the two implementations and the MatLAB prototype are shown in Table 8.2. All data of the test along with the non-averaged values can be found on the CD.

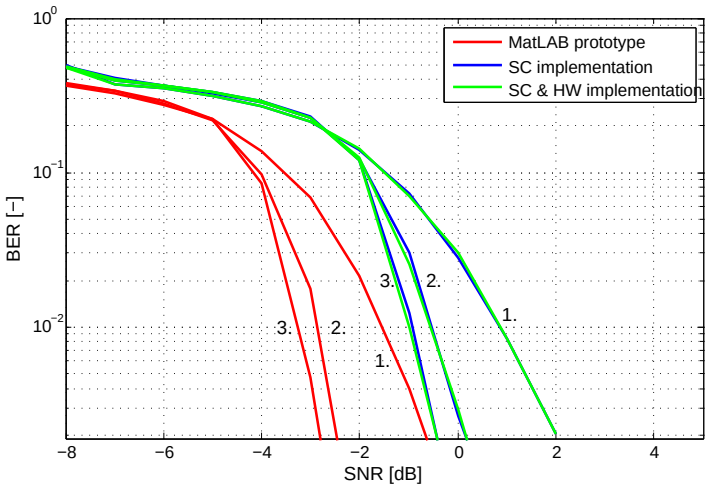


Figure 8.2: Results for the BER achieved at different levels of SNRs for the prototype in MatLAB and the two implementations. The numbers next to the curves indicate the number of iterations for the specific curve.

Test 2: Bit-error rate [-]									
	Soft-core			Soft-core & Hardware			MatLAB		
SNR [dB]	BER for iterations:								
	1	2	3	1	2	3	1	2	3
–8	0.4937	0.4919	0.4905	0.4758	0.4843	0.4798	0.3620	0.3735	0.3756
–7	0.3749	0.4074	0.3950	0.3730	0.3988	0.3957	0.3290	0.3410	0.3449
–6	0.3507	0.3660	0.3657	0.3519	0.3663	0.3678	0.2792	0.2844	0.2902
–5	0.3142	0.3280	0.3308	0.3154	0.3295	0.3342	0.2146	0.2082	0.2086
–4	0.2707	0.2873	0.2923	0.2695	0.2858	0.2933	0.1400	0.1021	0.0884
–3	0.2122	0.2265	0.2300	0.2153	0.2272	0.2277	0.0673	0.0185	0.0052
–2	0.1397	0.1218	0.1200	0.1423	0.1239	0.1200	0.0218	$7.5{\cdot}10^{-4}$	$1.9{\cdot}10^{-5}$
–1	0.0725	0.0305	0.0123	0.0701	0.0255	0.0099	0.0047	$9.8{\cdot}10^{-6}$	0
0	0.0283	0.0027	$4.9{\cdot}10^{-4}$	0.0304	0.0029	$6.1{\cdot}10^{-4}$	$3.6{\cdot}10^{-4}$	0	0
1	0.0084	$3.9{\cdot}10^{-4}$	$3.4{\cdot}10^{-4}$	0.0084	$2.4{\cdot}10^{-4}$	$2.2{\cdot}10^{-4}$	$2.4{\cdot}10^{-4}$	0	0
2	0.0020	$2.0{\cdot}10^{-4}$	$2.7{\cdot}10^{-4}$	0.0021	$1.7{\cdot}10^{-4}$	$1.5{\cdot}10^{-4}$	0	0	0

Table 8.2: Results of the second test, where the averaged BER is determined at 11 different SNRs, at three levels of iterations. The averaging is based on 20 repetitions for each SNR and for each number of iterations.

8.3 Intermediate conclusion

The results for the first test, regarding the execution time, shows that the combined soft-core and hardware implementation is faster than the pure soft-core implementation. Specifically, at for example 10 iterations, the two implementations differs by 2.614 s in favour of the hardware accelerated implementation. The results also show that the dependency between execution time and number of decoding iterations has an approximately linear relationship. In detail, the hardware accelerated implemtation is faster than the soft-core implementation by between 34.44% and 25.13% for one and 20 iterations, respectively.

The results illustrated in Figure 8.2 show that the curves of the two implementations have a shape similar to the curve for the prototype decoder. The BER decreases with increasing SNR and the number of decoding iterations. This suggests that the Turbo coding functionality is still preserved. At any SNR, the BERs of the implemented versions are roughly equal. However, compared to the prototype, the two implementations require SNRs 2 dB higher to achieve the same BER as the prototype.

From Table 8.2, it can be noticed that there is little to gain with regards to BER at low SNR. It is not until above -2 dB that the implementations really gain anything by more decoding iterations. From the same Table, it can be noticed that the BER of the two implementations starts to decrease at a faster rate from about -4 dB and above. This abrupt decrease starts around -5 dB for the MatLAB prototype. This can be explained by the higher precision of the prototype than of the embedded implementations.

Part III

Conclusion and Discussion

9 Conclusion

Channel coding is introduced in communication systems to reduce the information loss caused by noisy channels. The upper limit for the information rate is given by the Shannon limit [Shannon, 1948]. Turbo coding is capable of approaching this limit by using several decoders in an iterative manner.

This project investigated two implementations of Turbo coding on an FPGA using a soft-core processor and a soft-core processor with hardware acceleration. The objective of the project was to compare and evaluate the execution times for both implementations, in order to determine whether the hardware accelerated implementation had the lowest execution time.

A prototype of a Turbo coding scheme was implemented in MatLAB to verify the functionality and investigate the properties of the scheme. In this scheme, two recursive systematic convolutional encoders, concatenated in a parallel structure, along with a quadratic interleaver of length 2048, were employed.

Using an ideal noiseless channel, the encoding and decoding resulted in a bit error rate of zero. When the SNR decreased, the bit error rate increased as expected. For example, with two decoding iterations, a decrease in SNR from -2 to -3 dB resulted in an increase of the bit error rate from approximately $10^{-3.3}$ to $10^{-1.9}$. Furthermore, the bit error rate decreased with an increment in the number of decoding iterations. For example, at an SNR of -2 dB, the bit error rate decreased from $10^{-1.8}$ to $10^{-3.3}$ by performing two iterations instead of one.

The encoder and the channel emulation were kept in MatLAB on a PC, while the decoder was implemented in a Nios II soft-core processor on an Altera DE2 FPGA board. The encoded, noisy sequence was transmitted from MatLAB to the Nios II soft-core processor implementation. It was also made sure that the functionality and properties of the prototype were preserved when it was rewritten in the C programming language.

The soft-core implementation was profiled in order to determine which functions took up most of the execution time. The function for the Soft Output Viterbi Algorithm (SOVA) was found to be the most computationally intensive, so this function was further analysed using combined

data flow graphs and precedence graphs. Due to the dependencies between the SOVA function and the function for the Jacobian algorithm, the latter was also included in the analysis. From the analysis, it was observed that both the SOVA and Jacobian algorithm had inherent parallelism since several sequences of calculations did not depend on each other. Therefore, these sequences could be executed in parallel.

In order to exploit the inherent parallelism, a hardware accelerator was developed in VHDL to be placed as a peripheral alongside the Nios II on the FPGA. A large part of the contents of the loops in the SOVA algorithm was rewritten in VHDL so that the majority of the operations were moved to the accelerator, where they could be executed faster.

A comparison test was performed for both implementations in order to determine the execution times and compliance with the functionality of Turbo coding. The results for the test of compliance with the Turbo coding functionality showed slightly worse bit error rates in comparison to the previously tested prototype. At an SNR of -2 dB, the soft-core implementation and the implementation with a soft-core processor with hardware acceleration, showed a bit error rate of 0.1218 and 0.1239, respectively, whereas the prototype achieved a bit error rate of $7.4 \cdot 10^{-4}$. These results were achieved with the same number of decoding iterations.

However, since a resemblance between the prototype and the two implementations existed, these implementations were determined to comply with the Turbo coding functionality.

Regarding the project statement and determination of the execution time, test results showed that the hardware accelerated soft-core processor could perform decoding significantly faster than the soft-core processor alone. The execution time for 10 iterations was around 10.2 seconds for the pure soft-core processor and 7.5 seconds for the hardware aided implementation. The execution time was decreased by between 34 and 25 % when performing between 1 and 20 decoding iterations, respectively. Hence, it can be concluded that the hardware accelerated implementation is faster for the tested scenarios.

Implementing a hardware accelerator in combination with a soft-core processor, increased the area consumption by approximately 82 % compared to the implementation without hardware acceleration. In detail, the area consumption of the soft-core processor implementation took up 3,250 logical elements, while the hardware accelerated soft-core implementation took up 5,909 logical elements. Although the area consumption is not a hard constraint in this project, the hardware accelerated implementation fits on the selected FPGA.

Further considerations regarding the area consumption and execution time trade-offs are discussed in Chapter 10.

10 Discussion

As mentioned in Chapter 9, the addition of a hardware accelerator decreased the overall execution time by 25 to 34 %, but it also increased the area by 82 %. The need for a good trade-off between area consumption and execution time is clear from these results. In some applications it would not be tolerable to double the area by adding a hardware accelerator as done in this project, whereas other applications would require a larger decrease in the execution time. Each application should therefore be investigated in order to find the right trade-off between execution time and area consumption.

The hardware accelerator, designed in this project, is exploiting the concurrency and speed provided by the FPGA to decrease the time spent on each branch cost computation. This results in the fact that the hardware accelerator computes the branch cost in under two clock cycles. Other implementation structures could have been chosen, such as the Finite State Machine (FSM) with datapath. This structure uses an FSM to control a set of functional units capable of doing the operations needed to calculate the branch cost. This approach offers more control and the possibility of reusing functional units, which in turn reduces the area consumption. Using the FSM approach will furthermore make it possible to control the trade-off between area and execution time with a higher degree of granularity.

Even after implementing the hardware acceleration, the software was still responsible for most of the execution time. This indicates that a larger part of the SOVA algorithm could, with advantage, be implemented in the hardware accelerator. However, such a decision might not only increase the consumed area, but also increase the amount of communication required between the soft-core processor and the hardware accelerator.

In this project, the approach for delegating computations to the hardware accelerator was primarily based on which parts were the most computationally intensive. However, the amount of communication with the hardware accelerator could also have been factored in. An analysis of this would most likely have shown, that a smaller amount of communication would have been required if the loops in the Soft Output Viterbi Algorithm (SOVA) algorithm were also moved to the hardware accelerator and not only the inner parts of the loops. However, a large amount

of communication would, instead, be required between the hardware accelerator and the memory circuits. Furthermore, such a decision would cause an increase in both area consumption and duration of the design process.

The precision of the Turbo decoder was not affected by the addition of the hardware accelerator to the soft-core processor. However, both implementations perform worse than the MatLAB prototype. This was illustrated by Figure 8.2 on page 83, which showed that the two implementations required 2 dB higher SNR to achieve the same BER.

This is assumed to be caused by the fact that the two implementations were done with a fixed point representation, whereas the prototype did computations in floating point. The two implementations could have utilised floating point representation. However, that would have significantly increased the execution time and area consumption.

Part IV

Appendix

A

Derivation of the extrinsic LLR

The purpose of this appendix is to derive the extrinsic Log Likelihood Ratio (LLR) from the following expression for the normal LLR:

$$\begin{aligned} L_t(\underline{\hat{\mathbf{x}}}) &= \mu_{F,t}(\underline{\mathbf{P}}^0_{\min}) + \nu(t, S_t(\underline{\mathbf{P}}^0_{\min}), 0) + \mu_{B,t+1}(\underline{\mathbf{P}}^0_{\min}) \\ &- \left(\mu_{F,t}(\underline{\mathbf{P}}^1_{\min}) + \nu(t, S_t(\underline{\mathbf{P}}^1_{\min}), 1) + \mu_{B,t+1}(\underline{\mathbf{P}}^1_{\min})\right) \end{aligned} \tag{A.1}$$

This Appendix is based on knowledge obtained from [Vucetic and Yuan, 2000].

The extrinsic LLR should contain all the information from the decoding process which isn't directly linked to the systematic bit (i.e. the first bit in each received bit-string). We therefore focus on the part of the expression above in which the term $\hat{y}_{t,1}$ is present.

$$\begin{aligned} \nu(t, S_t(\underline{\mathbf{P}}^0_{\min}), 0) &- \nu(t, S_t(\underline{\mathbf{P}}^1_{\min}), 1) \\ &= \sum_{i=1}^{n} (\hat{y}_{t,i} - y_{S_t(P^0_{\min}),0,i})^2 - (\hat{y}_{t,i} - y_{S_t(P^1_{\min}),1,i})^2 \\ &+ \ln(\mathrm{P}(x_t = 1)) - \ln(\mathrm{P}(x_t = 0)) \end{aligned} \tag{A.2}$$

$$\begin{aligned} &= (\hat{y}_{t,1} - y_{S_t(P^0_{\min}),0,1})^2 - (\hat{y}_{t,1} - y_{S_t(P^1_{\min}),1,1})^2 \\ &+ \sum_{i=2}^{n} (\hat{y}_{t,i} - y_{S_t(P^0_{\min}),0,i})^2 - (\hat{y}_{t,i} - y_{S_t(P^1_{\min}),1,i})^2 \\ &+ \ln\left(\frac{\mathrm{P}(x_t = 1)}{\mathrm{P}(x_t = 0)}\right) \end{aligned} \tag{A.3}$$

$$
\begin{aligned}
&= \hat{y}_{t,1}^2 - \hat{y}_{t,1}^2 + y_{S_t(P^0_{\min}),0,1}^2 - y_{S_t(P^1_{\min}),1,1}^2 \\
&- 2\hat{y}_{t,1} \cdot y_{S_t(P^0_{\min}),0,1} + 2\hat{y}_{t,1} \cdot y_{S_t(P^1_{\min}),1,1} \\
&+ \sum_{i=2}^{n} (\hat{y}_{t,i} - y_{S_t(P^0_{\min}),0,i})^2 - (\hat{y}_{t,i} - y_{S_t(P^1_{\min}),1,i})^2 \\
&+ \ln\left(\frac{\mathrm{P}(x_t = 1)}{\mathrm{P}(x_t = 0)}\right)
\end{aligned} \tag{A.4}
$$

It is known that $y_{S_{t-1}(P^0_{\min}),0,1} = -1$ and $y_{S_{t-1}(P^1_{\min}),1,1} = 1$ since the encoder is systematic, and the first bit in each received bit-string therefore must be the input-bit.

$$
\begin{aligned}
&= \hat{y}_{t,1} \cdot 4 + \sum_{i=2}^{n} (\hat{y}_{t,i} - y_{S_t(P^0_{\min}),0,i})^2 - (\hat{y}_{t,i} - y_{S_t(P^1_{\min}),1,i})^2 \\
&+ \ln\left(\frac{\mathrm{P}(x_t = 1)}{\mathrm{P}(x_t = 0)}\right)
\end{aligned} \tag{A.5}
$$

The extrinsic LLR can then be defined to be:

$$
\begin{aligned}
L_e(\underline{\hat{\mathbf{x}}}) &= \mu_{F,t}(\underline{\mathbf{P}}^0_{\min}) + \mu_{B,t+1}(\underline{\mathbf{P}}^0_{\min}) - \mu_{F,t}(\underline{\mathbf{P}}^1_{\min}) - \mu_{B,t}(\underline{\mathbf{P}}^1_{\min}) \\
&+ \sum_{i=2}^{n} (\hat{y}_{t,i} - y_{S_t(\underline{\mathbf{P}}^0_{\min}),0,1})^2 - (\hat{y}_{t,i} - y_{S_t(\underline{\mathbf{P}}^1_{\min}),1,i})^2
\end{aligned} \tag{A.6}
$$

Which results in the following expression for the normal LLR:

$$
L_t(\underline{\hat{\mathbf{x}}}) = \hat{y}_{t,1} \cdot 4 + L_e(\underline{\hat{\mathbf{x}}}) + La(\underline{\mathbf{x}}) \tag{A.7}
$$

B

Software for Nios II soft-core processor

B.1 turbo_coding.h

```
1  #ifndef _TURBO_CODING_H
2  #define _TURBO_CODING_H
3  #include <stdarg.h>
4  #include <stdio.h>
5  #include <unistd.h>
6  //#define NIOS 123
7  #ifndef NIOS //in executed on the PC
8    #include <time.h>
9  #endif
10
11 /* Definitions */
12 #define BLOCK_SIZE 2048
13 #define ALTERA_AVALON_UART_USE_IOCTL TRUE
14 #define UART1_NAME uart
15
16
17 /* Function Headers */
18
19 void decoder(char fixed_it);
20 void interleave(int LUT[], int vector[]);
21 void deinterleave(int LUT[], int vector[]);
22 void sova(int sys[], int par1[], int par2[]);
23
24 int jacob_log(int x);
25
26 /* Global Variables */
27 #ifndef NIOS //if executed on PC
28 FILE *outfile;
29 FILE *infile;
30 #endif
31
32 char dec_data[BLOCK_SIZE];  // Array for the decoded data
33 int sys[BLOCK_SIZE];    // Array for systematic bits
34 int sys_int[BLOCK_SIZE];
35 int par11[BLOCK_SIZE+2];    // Arrays for parity bits.
36 int par12[BLOCK_SIZE+2];
37 int par21[BLOCK_SIZE+2];
38 int par22[BLOCK_SIZE+2];
39
```

```
40 int LLR[BLOCK_SIZE]; // Soft output
41 int La[BLOCK_SIZE];  // A priori values (LLR)
42 #endif
```

B.2 main.c

```
1  #include "turbo_coding.h"
2  #include "system.h"
3  #include <fcntl.h>
4  #include "alt_types.h"
5  #include "sys/alt_timestamp.h"
6  #include "sys/alt_alarm.h"
7
8  int main(void) {
9
10   FILE *fp;
11   alt_u16 proc_u16_divisor;
12   alt_u32 time_start_nios, time_elapsed_nios, ticks_per_second;
13   unsigned char iterations, char1, char2, char3, char4,char_sign;
14   int passed_time, i;
15   int sys_length, par11_length, par12_length, par21_length, par22_length;
16   char* output;
17
18   #ifdef NIOS /*If not running on the NIOS platform*/
19     printf("Hello from Nios II , i am inside nios!\n");
20     fp = fopen(UART_0_NAME,"r+");
21
22     if(fp) {
23       printf("UART open. \n");
24       fprintf(fp, "UART opens\r\n");
25     } else {
26       printf("UART can't opened. \n");
27     }
28
29   #else /* Else connect the file-pointers to files on the PC */
30
31   outfile = fopen( "output.txt", "w" ); // Open the file for write if not on the nios
       processor
32   infile = fopen( "input.txt", "r" ); // Open the file for read if not on the nios
       processor
33
34   #endif /* NIOS */
35
36   /*************************************************************************/
37   /**                  Receive data from the file or MatLAB                **/
38   /*************************************************************************/
39
40   /*  reading data and storing into memory.  51251 bytes of data will be send from
41    * MatLAB to nios */
42
43   #ifdef NIOS /*If  running on the NIOS platform*/
44
45     /* Read the number of iterations that should be runned from the UART */
46
47     iterations = fgetc(fp);
48     printf("number of iteration: %d",iterations);
49
50     /* Read the length of the systematic bit sequence from the UART */
51
52     char1 = fgetc(fp);
53     char2 = fgetc(fp);
54     sys_length = char1<<8;
55     sys_length += char2;
56
57     /* Read the length of the first parity bit sequence first encoder from the UART */
58
59     char1 = fgetc(fp);
60     char2 = fgetc(fp);
```

```
61      par11_length = char1<<8;
62      par11_length += char2;
63
64      /* Read the length of the second parity bit sequence first encoder from the UART */
65
66      char1 = fgetc(fp);
67      char2 = fgetc(fp);
68      par12_length = char1<<8;
69      par12_length += char2;
70
71      /* Read the length of the first parity bit sequence second encoder from the UART */
72
73      char1 = fgetc(fp);
74      char2 = fgetc(fp);
75      par21_length = char1<<8;
76      par21_length += char2;
77
78      /* Read the length of the second parity bit sequence second encoder from the UART */
79
80      char1 = fgetc(fp);
81      char2 = fgetc(fp);
82      par22_length = char1<<8;
83      par22_length += char2;
84
85      /* Receive the systematic bit sequence (send as 4 bytes + 1 sign byte) */
86
87      for (i=0 ; i<sys_length ; i++) {
88        char_sign = fgetc(fp);
89        char1 = fgetc(fp);
90        char2 = fgetc(fp);
91        char3 = fgetc(fp);
92        char4 = fgetc(fp);
93        sys[i] = char1<<24;
94        sys[i] += char2<<16;
95        sys[i] += char3<<8;
96        sys[i] += char4;
97        if (char_sign==1) {
98          sys[i] = sys[i]*-1;
99        }
100     }
101
102     /* Receive the first parity bit sequence first encoder (send as 4 bytes + 1 sign byte
          ) */
103
104     for (i=0; i<par11_length; i++) {
105       char_sign = fgetc(fp);
106       char1 = fgetc(fp);
107       char2 = fgetc(fp);
108       char3 = fgetc(fp);
109       char4 = fgetc(fp);
110       par11[i] = char1<<24;
111       par11[i] += char2<<16;
112       par11[i] += char3<<8;
113       par11[i] += char4;
114       if (char_sign==1) {
115         par11[i] = par11[i]*-1;
116       }
117     }
118
119     /* Receive the second parity bit sequence first encoder (send as 4 bytes + 1 sign
          byte) */
120
121     for (i=0; i<par12_length; i++) {
122       char_sign = fgetc(fp);
123       char1 = fgetc(fp);
124       char2 = fgetc(fp);
125       char3 = fgetc(fp);
126       char4 = fgetc(fp);
127       par12[i] = char1<<24;
128       par12[i] += char2<<16;
```

```
129         par12[i] += char3<<8;
130         par12[i] += char4;
131         if (char_sign==1) {
132           par12[i] = par12[i]*-1;
133         }
134       }
135
136       /* Receive the first parity bit sequence second encoder (send as 4 bytes + 1 sign
              byte) */
137
138       for (i=0; i<par21_length; i++) {
139         char_sign = fgetc(fp);
140         char1 = fgetc(fp);
141         char2 = fgetc(fp);
142         char3 = fgetc(fp);
143         char4 = fgetc(fp);
144         par21[i] = char1<<24;
145         par21[i] += char2<<16;
146         par21[i] += char3<<8;
147         par21[i] += char4;
148         if (char_sign==1) {
149           par21[i] = par21[i]*-1;
150         }
151       }
152
153       /* Receive the second parity bit sequence second encoder (send as 4 bytes + 1 sign
              byte) */
154
155       for (i=0; i<par22_length; i++) {
156         char_sign = fgetc(fp);
157         char1 = fgetc(fp);
158         char2 = fgetc(fp);
159         char3 = fgetc(fp);
160         char4 = fgetc(fp);
161         par22[i] = char1<<24;
162         par22[i] += char2<<16;
163         par22[i] += char3<<8;
164         par22[i] += char4;
165         if (char_sign==1) {
166           par22[i] = par22[i]*-1;
167         }
168       }
169
170
171     #else  /* If the code is not executed on the nios, the same data should be retrieved,
                  but from a file */
172
173       iterations = getc(infile);
174
175       char1 = getc(infile);
176       char2 = getc(infile);
177       sys_length = char1<<8;
178       sys_length += char2;
179
180       char1 = getc(infile);
181       char2 = getc(infile);
182       par11_length = char1<<8;
183       par11_length += char2;
184
185       char1 = getc(infile);
186       char2 = getc(infile);
187       par12_length = char1<<8;
188       par12_length += char2;
189
190       char1 = getc(infile);
191       char2 = getc(infile);
192       par21_length = char1<<8;
193       par21_length += char2;
194
195       char1 = getc(infile);
```

```
196       char2 = getc(infile);
197       par22_length = char1<<8;
198       par22_length += char2;
199
200       for (i=0 ; i<sys_length ; i++) {
201         char_sign = getc(infile);
202         char1 = getc(infile);
203         char2 = getc(infile);
204         char3 = getc(infile);
205         char4 = getc(infile);
206         sys[i] = char1<<24;
207         sys[i] += char2<<16;
208         sys[i] += char3<<8;
209         sys[i] += char4;
210         if (char_sign==1) {
211           sys[i] = sys[i]*-1;
212         }
213       }
214
215       for (i=0; i<par11_length; i++) {
216         char_sign = getc(infile);
217         char1 = getc(infile);
218         char2 = getc(infile);
219         char3 = getc(infile);
220         char4 = getc(infile);
221         par11[i] = char1<<24;
222         par11[i] += char2<<16;
223         par11[i] += char3<<8;
224         par11[i] += char4;
225         if (char_sign==1) {
226           par11[i] = par11[i]*-1;
227         }
228       }
229
230       for (i=0; i<par12_length; i++) {
231         char_sign = getc(infile);
232         char1 = getc(infile);
233         char2 = getc(infile);
234         char3 = getc(infile);
235         char4 = getc(infile);
236         par12[i] = char1<<24;
237         par12[i] += char2<<16;
238         par12[i] += char3<<8;
239         par12[i] += char4;
240         if (char_sign==1) {
241           par12[i] = par12[i]*-1;
242         }
243       }
244
245       for (i=0; i<par21_length; i++) {
246         char_sign = getc(infile);
247         char1 = getc(infile);
248         char2 = getc(infile);
249         char3 = getc(infile);
250         char4 = getc(infile);
251         par21[i] = char1<<24;
252         par21[i] += char2<<16;
253         par21[i] += char3<<8;
254         par21[i] += char4;
255         if (char_sign==1) {
256           par21[i] = par21[i]*-1;
257         }
258       }
259
260       for (i=0; i<par22_length; i++) {
261         char_sign = getc(infile);
262         char1 = getc(infile);
263         char2 = getc(infile);
264         char3 = getc(infile);
265         char4 = getc(infile);
```

```
266       par22[i] = char1<<24;
267       par22[i] += char2<<16;
268       par22[i] += char3<<8;
269       par22[i] += char4;
270       if (char_sign==1) {
271         par22[i] = par22[i]*-1;
272       }
273     }
274
275   #endif /* NIOS */
276
277   /*************************************************/
278   /**                Start Timer                  **/
279   /*************************************************/
280
281   #ifdef NIOS  /* If executed on nios */
282
283     ticks_per_second = alt_ticks_per_second();   // Test if the timer is setup correctly
284     if (ticks_per_second == 0){
285       printf("timer hardware not works well\n");
286       return (0);
287     }
288     time_start_nios = alt_nticks();   //Save the timer value
289
290   #else /* If executed on PC */
291
292     time_t time_start, time_stop;
293     time(&time_start);
294
295   #endif //NIOS
296
297   /*****************************************************************************/
298   /**               Run the decoder on the data                               **/
299   /*****************************************************************************/
300
301   decoder(iterations);
302
303   /*************************************************/
304   /**                   Stop Timer                **/
305   /*************************************************/
306
307   #ifdef NIOS  /* If executed on nios */
308
309     time_elapsed_nios = alt_nticks() - time_start_nios;  //calculate the time used to
          decode
310     printf(" time duration :%.3f seconds\n",(float)time_elapsed_nios/(float)
          ticks_per_second);
311     printf("ticks_per_second %d",ticks_per_second);
312
313   #else  /* If executed on PC */
314
315     time(&time_stop); //Save the timer-value
316     passed_time=(int) time_stop-time_start; // calculate the time used to decode
317
318   #endif //NIOS
319
320   /*****************************************************************************/
321   /**      Send the decoded data to MatLAB (or write to file)                 **/
322   /*****************************************************************************/
323
324   #ifdef NIOS  /* If executed on nios */
325
326     char1 = time_elapsed_nios >>24;
327     char2 = time_elapsed_nios >>16;
328     char3 = time_elapsed_nios >>8;
329     char4 = time_elapsed_nios ;
330
331     fprintf(fp, "%c",char1);
332     fprintf(fp, "%c",char2);
333     fprintf(fp, "%c",char3);
```

```
334       fprintf(fp, "%c",char4);
335
336     #else  /* If executed on PC */
337
338       char1 = passed_time>>24;
339       char2 = passed_time>>16;
340       char3 = passed_time>>8;
341       char4 = passed_time;
342
343       putc(char1,outfile);
344       putc(char2,outfile);
345       putc(char3,outfile);
346       putc(char4,outfile);
347
348     #endif //NIOS
349
350
351     #ifdef NIOS  /* If executed on nios */
352
353       for(i=0; i<BLOCK_SIZE; i++) {
354         fprintf(fp, "%c",dec_data[i]);
355         printf("%d ,%c",i,dec_data[i]);
356       }
357
358     #else  /* If executed on PC */
359
360       for(i=0; i<BLOCK_SIZE; i++) {
361         putc(dec_data[i],outfile);
362       }
363
364     #endif //NIOS
365
366     return 0;
367   }
```

B.3 decoder.c

```
1   #include "turbo_coding.h"
2
3   void decoder(char fixed_it) { // *dec_data
4
5     /* Definitions */
6
7     // Here the variable LUT[] should be defined, but as it is a large array it is omitted
        to save space.
8
9     int i=0;  // Counting variable
10
11    /* Assigning values */
12
13    for (i=0 ; i<BLOCK_SIZE ; i++) {  // A priori values start at 0
14      La[i] = 0;  // as 1 and 0 are equally likely
15    }
16
17    for (i=0 ; i<BLOCK_SIZE ; i++) {
18      sys_int[i] = sys[i];
19    }
20
21    interleave(LUT,sys_int);
22
23    /* Iterative turbo coding */
24
25    for (i=0 ; i<fixed_it ; i++) {  // Run for a fixed # of iterations
26      sova(sys,par11,par12);  // DEC1 with SOVA alg. is called
27      interleave(LUT,La);          // Ext. values from DEC1 are interleaved
28      sova(sys_int,par21,par22);  // DEC2 is called
29      deinterleave(LUT,La);          // Ext. values from DEC2 are deinterleaved
30    }
```

```
31
32    deinterleave(LUT,LLR);      // Final estimation from DEC2 is deinterleaved
33
34    /* Hard decision output */
35    for (i=0; i<BLOCK_SIZE ; i++) {
36      if (LLR[i]>0)
37        dec_data[i] = 1;
38      else
39        dec_data[i] = 0;
40    }
41
42
43  }
```

B.4 sova.c

```
1   #include "turbo_coding.h"
2
3   #define LARGE_NUMBER 20000000
4   #define NSTATES 4
5
6   void sova(int sys_internal[], int par1[], int par2[]) {
7     int i,t;
8
9     char outputindex0, outputindex1;
10    int outpar1,outpar2, branch0, branch1, logprob, temp_cost, La_temp;
11    static long temp,syscost,par1cost,par2cost;
12
13    //Array structure containing the output from the encoder given all possible states and
        inputs
14    static const int output[8][2] = {{65536,-65536}, {65536,65536}, {-65536,65536},
        {-65536,-65536}, {-65536,-65536}, {-65536,65536}, {65536,65536}, {65536,-65536}};
15
16    //Arrays containing the previous or next state given the input
17    static const char transitions[4][2] = {{0,1},{3,2},{1,0},{2,3}};
18    static const char transitions_back[4][2] = {{0,2},{2,0},{3,1},{1,3}};
19
20    //Large array elements to contain the forward and backward path costs
21    static int mu_best[NSTATES][BLOCK_SIZE+3];
22    static int mu_best_back[NSTATES][BLOCK_SIZE+3];
23
24    int mu0 = 0;
25    int mu1 = 0;
26
27    //Initial cost values
28    mu_best[0][0] = 0; //zero cost of starting in point 0,0
29    mu_best_back[0][BLOCK_SIZE+2] = 0; // and ending in point 0,BLOCK_SIZE
30
31    for (i=1; i<NSTATES; i++) {
32      mu_best[i][0] = 65536000;  //1000 in cost for not starting in 0,0
33      mu_best_back[i][BLOCK_SIZE+2] = 65536000; //and not ending in 0,BLOCK_SIZE
34    }
35
36
37    /*forward run*/
38
39    for (t=1;t<=BLOCK_SIZE+2;t++) {        //Visit all points in the grid (that is all
        states and all bits)
40      for (i=0; i<NSTATES; i++) {
41
42        if (t > BLOCK_SIZE) { //If we are considering the termination bits (last two bits
            in the parity sequence) the systematic bit should be set to zero
43          temp = 65536;
44        } else {
45          temp = sys_internal[t-1] + 65536;
46        }
47        temp = temp >> 8;
48        syscost = temp*temp; //calculate the cost related to the systematic bit if the
```

```
        reference is 0
49
50      //Calculate the output of the encoder given a 0
51      outputindex0 = transitions[i][0];
52      outpar1 = output[outputindex0+NSTATES][0];
53      outpar2 = output[outputindex0+NSTATES][1];
54
55      //calculate cost related to the first parity bit
56      temp = par1[t-1] - outpar1;
57      temp = temp >> 8;
58      par1cost = temp*temp;
59
60      //calculate cost related to the second parity bit
61      temp = par2[t-1] - outpar2;
62      temp = temp >> 8;
63      par2cost = temp*temp;
64
65      if (t-1 > BLOCK_SIZE) {
66        La_temp = 0;
67      } else {
68        La_temp = La[t-1];
69      }
70      //calculate the cost related to the a priori probability
71      logprob = -1*jacob_log(La_temp);
72
73      branch0 = syscost + par1cost + par2cost - logprob; //Calculate the full branch cost
74
75      if (t > BLOCK_SIZE) {
76        temp = -65536;
77      } else {
78        temp = sys_internal[t-1] - 65536;
79      }
80      temp = temp >> 8;
81      syscost = temp*temp; //calculate the cost related to the systematic bit if the
            reference is 1
82
83      //calculate the output of the encoder given a 1
84      outputindex1 = transitions[i][1];
85      outpar1 = output[outputindex1][0];
86      outpar2 = output[outputindex1][1];
87
88      //calculate cost related to the first parity bit
89      temp = par1[t-1] - outpar1;
90      temp = temp >> 8;
91      par1cost = temp*temp;
92
93      //calculate cost related to the second parity bit
94      temp = par2[t-1] - outpar2;
95      temp = temp >> 8;
96      par2cost = temp*temp;
97
98      if (t-1 > BLOCK_SIZE) {
99        La_temp = 0;
100     } else {
101       La_temp = La[t-1];
102     }
103     //calculate the cost related to the a priori probability
104     logprob = La_temp-jacob_log(La_temp);
105
106     branch1 = syscost + par1cost + par2cost - logprob; //calculate the full branch cost
107
108     branch0 = mu_best[outputindex0][t-1] + branch0; //calculate the path cost given a 0
109
110     if(branch0 < mu_best[outputindex0][t-1]) { //check for overflow
111       branch0 = LARGE_NUMBER;
112     }
113
114     branch1 = mu_best[outputindex1][t-1] + branch1; //calculate the path cost given a 1
115
116     if(branch1 < mu_best[outputindex1][t-1]) { //check for overflow
```

```
117           branch1 = LARGE_NUMBER;
118         }
119
120         if ( branch0 <= branch1) { //Set the path cost in this point to the lowest of the
              two path costs
121           mu_best[i][t] = branch0;
122         } else {
123           mu_best[i][t] = branch1;
124         }
125       }
126     }
127
128     /* Backwards Run - The exact same as the forward run, just other indices */
129
130     for (t=BLOCK_SIZE+1; t>=0; t--) {
131       for (i=0; i< NSTATES; i++) {
132         if (t >= BLOCK_SIZE) {
133           temp = 65536;
134         } else {
135           temp = sys_internal[t] + 65536;
136         }
137         temp = temp >> 8;
138         syscost = temp*temp;
139
140         outpar1 = output[i+NSTATES][0];
141         outpar2 = output[i+NSTATES][1];
142
143         temp = par1[t] - outpar1;
144         temp = temp >> 8;
145         par1cost = temp*temp;
146
147         temp = par2[t] - outpar2;
148         temp = temp >> 8;
149         par2cost = temp*temp;
150
151         if (t >= BLOCK_SIZE) {
152           La_temp = 0;
153         } else {
154           La_temp = La[t];
155         }
156         logprob = -1*jacob_log(La_temp);
157
158         branch0 = syscost + par1cost + par2cost - logprob;
159         if (t > BLOCK_SIZE) {
160           temp = -65526;
161         } else {
162           temp = sys_internal[t] - 65536;
163         }
164         temp = temp >> 8;
165         syscost = temp*temp;
166
167         outpar1 = output[i][0];
168         outpar2 = output[i][1];
169
170         temp = par1[t] - outpar1;
171         temp = temp >> 8;
172         par1cost = temp*temp;
173
174         temp = par2[t] - outpar2;
175         temp = temp >> 8;
176         par2cost = temp*temp;
177
178         if (t >= BLOCK_SIZE) {
179           La_temp = 0;
180         } else {
181           La_temp = La[t];
182         }
183         logprob = La_temp-jacob_log(La_temp);
184
185         branch1 = syscost + par1cost + par2cost - logprob;
```

```
186
187         branch0 = mu_best_back[transitions_back[i][0]][t+1] + branch0;
188         if(branch0 < mu_best_back[transitions_back[i][0]][t+1]) {
189           branch0 = LARGE_NUMBER;
190         }
191         branch1 = mu_best_back[transitions_back[i][1]][t+1] + branch1;
192         if(branch1 < mu_best_back[transitions_back[i][1]][t+1]) {
193           branch1 = LARGE_NUMBER;
194         }
195
196         if ( branch0 <= branch1) {
197           mu_best_back[i][t] = branch0;
198         } else {
199           mu_best_back[i][t] = branch1;
200         }
201       }
202     }
203
204     /* Calculation of soft-output and extrinsic information */
205
206     for (t=0;t<BLOCK_SIZE;t++) { // Visit all bits
207       for (i=0; i < NSTATES; i++) { //and all states
208
209         if (t-1 > BLOCK_SIZE) {
210           La_temp = 0;
211         } else {
212           La_temp = La[t];
213         }
214
215         //calculate the branch cost given a 0
216         logprob = -1*jacob_log(La_temp);
217
218         temp = sys_internal[t] + 65536;
219         temp = temp >> 8;
220         syscost = temp*temp;
221
222         outputindex0 = transitions_back[i][0];
223         outpar1 = output[outputindex0+NSTATES][0];
224         outpar2 = output[outputindex0+NSTATES][1];
225
226         temp = par1[t] - outpar1;
227         temp = temp >> 8;
228         par1cost = temp*temp;
229
230         temp = par2[t] - outpar2;
231         temp = temp >> 8;
232         par2cost = temp*temp;
233
234         branch0 = syscost + par1cost + par2cost - logprob;
235
236         //Calculate the full cost of a path through this point and with a 0 in this branch
237         temp_cost = mu_best[i][t] + mu_best_back[outputindex0][t+1] + branch0;
238         if(((t <= 1024) && (temp_cost < mu_best[i][t])) || ((t > 1024) && (temp_cost <
              mu_best_back[outputindex0][t+1]))) { //check for overflow
239           temp_cost = LARGE_NUMBER;
240         }
241
242         if (temp_cost < mu0 || i == 0) { //if the calculated cost are smaller than the
              other costs calculated for a 0 branch
243           mu0 = temp_cost;          //Save it
244         }
245
246         //calculate the branch cost given a 1
247         if (t-1 > BLOCK_SIZE) {
248           La_temp = 0;
249         } else {
250           La_temp = La[t];
251         }
252         logprob = La_temp-jacob_log(La_temp);
253
```

```
254         temp = sys_internal[t] - 65536;
255         temp = temp >> 8;
256         syscost = temp*temp;
257
258         outputindex1 = transitions_back[i][1];
259         outpar1 = output[outputindex1][0];
260         outpar2 = output[outputindex1][1];
261
262         temp = par1[t] - outpar1;
263         temp = temp >> 8;
264         par1cost = temp*temp;
265
266         temp = par2[t] - outpar2;
267         temp = temp >> 8;
268         par2cost = temp*temp;
269
270         branch1 = syscost + par1cost + par2cost - logprob;
271
272         //calculate the full cost of a path through this point and with a 1 in this branch
273         temp_cost = mu_best[i][t] + mu_best_back[outputindex1][t+1] + branch1;
274         if(((t <= 1024) && (temp_cost < mu_best[i][t])) || ((t > 1024) && (temp_cost <
               mu_best_back[outputindex1][t+1]))) { //check for overflow
275           temp_cost = LARGE_NUMBER;
276         }
277
278         if (temp_cost < mu1 || i==0) { //if the calculated cost are smaller than the other
               costs calculated for a 1 branch
279           mu1 = temp_cost;        //Save is
280         }
281
282       }
283
284       LLR[t] = mu0 - mu1; //Calculate the soft output
285       La[t] = LLR[t] - 4*sys_internal[t] - La[t]; //calculate the extrinsic information
286     }
287 }
```

B.5 hw_sova.c

```
1  #include "turbo_coding.h"
2
3  #define LARGE_NUMBER 20000000
4  #define NSTATES 4
5
6  //define base address of hw-accelerator
7  #define base1 0x00101000
8
9  //define macros for communicating with the hw-accelerator
10 #define WR_SOVA1(offset,data) IOWR_32DIRECT(base1,offset,data)
11 #define RD_SOVA1(offset) IORD_32DIRECT(base1,offset)
12
13 //define offset adresses for the variables
14 #define sysrec 0x00
15 #define sysref 0x04
16 #define par1rec 0x08
17 #define par1ref 0x0C
18 #define par2rec 0x10
19 #define par2ref 0x14
20 #define log_in 0x18
21 #define branchres 0x1C
22
23
24
25 void sova(int sys_internal[], int par1[], int par2[]) {
26   int i,t;
27   int count;
28       int brA=-2147483648; //Largest number to represent in HW
29   int outputindex0, outputindex1;
```

```
30  int outpar1,outpar2, branch0, branch1, logprob, temp_cost, La_temp;
31  static long temp,syscost,par1cost,par2cost;
32
33  //Array structure containing the output from the encoder given all possible states and
       inputs
34  static const int output[8][2] = {{65536,-65536}, {65536,65536}, {-65536,65536},
       {-65536,-65536}, {-65536,-65536}, {-65536,65536}, {65536,65536}, {65536,-65536}};
35
36  //Arrays containing the previous or next state given the input
37  static const int transitions[4][2] = {{0,1},{3,2},{1,0},{2,3}};
38  static const int transitions_back[4][2] = {{0,2},{2,0},{3,1},{1,3}};
39
40  //Large array elements to contain the forward and backward path costs
41  static int mu_best[NSTATES][BLOCK_SIZE+3];
42  static int mu_best_back[NSTATES][BLOCK_SIZE+3];
43
44  int mu0 = 0;
45  int mu1 = 0;
46
47  //Initial cost values
48  mu_best[0][0] = 0; //zero cost of starting in point 0,0
49  mu_best_back[0][BLOCK_SIZE+2] = 0; // and ending in point 0,BLOCK_SIZE
50
51  for (i=1; i<NSTATES; i++) {
52    mu_best[i][0] = 65536000;  //1000 in cost for not starting in 0,0
53    mu_best_back[i][BLOCK_SIZE+2] = 65536000; //and not ending in 0,BLOCK_SIZE
54  }
55
56
57
58
59  /*forward run*/
60
61  for (t=1;t<=BLOCK_SIZE+2;t++) {
62    for (i=0; i<NSTATES; i++) {
63      //Send the systematic information to the hardware (sysref = 0)
64      if (t > BLOCK_SIZE) {
65        WR_SOVA1(sysref,65536);
66        WR_SOVA1(sysrec,-65536);
67      } else {
68        WR_SOVA1(sysrec,sys_internal[t-1]);
69        WR_SOVA1(sysref,-65536);
70      }
71
72      //Calculate the output of the encoder given a 0
73      outputindex0 = transitions[i][0];
74      outpar1 = output[outputindex0+NSTATES][0];
75            outpar2 = output[outputindex0+NSTATES][1];
76
77      //send the par1 information to the hardware
78      WR_SOVA1(par1rec,outpar1);
79      WR_SOVA1(par1ref,par1[t-1]);
80
81      //send the par2 information to the hardware
82      WR_SOVA1(par2rec,outpar2);
83      WR_SOVA1(par2ref,par2[t-1]);
84
85      //Send the information related to the logprob to hardware
86      if (t-1 > BLOCK_SIZE) {
87        WR_SOVA1(log_in,0);
88      } else {
89        WR_SOVA1(log_in,La[t-1]);
90      }
91
92      //Read the branch cost
93      branch0 = RD_SOVA1(branchres);
94
95      //Send the systematic information to the hardware (sysref = 1)
96      if (t > BLOCK_SIZE) {
97        WR_SOVA1(sysref,65536);
```

```
98          WR_SOVA1(sysrec,0);
99        } else {
100         WR_SOVA1(sysrec,sys_internal[t-1]);
101         WR_SOVA1(sysref,65536);
102       }
103
104       //Calculate the output of the encoder given a 0
105       outputindex1 = transitions[i][1];
106       outpar1 = output[outputindex1][0];
107             outpar2 = output[outputindex1][1];
108
109       //send the par1 information to the hardware
110       WR_SOVA1(par1rec,outpar1);
111       WR_SOVA1(par1ref,par1[t-1]);
112
113       //send the par2 information to the hardware
114       WR_SOVA1(par2rec,outpar2);
115       WR_SOVA1(par2ref,par2[t-1]);
116
117       //Send the information related to the logprob to hardware
118       if (t-1 > BLOCK_SIZE) {
119           WR_SOVA1(log_in,0); // La_temp = 0-;
120       } else {
121         WR_SOVA1(log_in,La[t-1]); //La_temp = La[t-1];
122       }
123
124       //Read the branch cost
125       branch1 = RD_SOVA1(branchres);
126
127       branch0 = mu_best[outputindex0][t-1] + branch0;//calculate the path cost given a 0
128       if(branch0 < mu_best[outputindex0][t-1]) { //check for overflow
129         branch0 = LARGE_NUMBER;
130       }
131
132       branch1 = mu_best[outputindex1][t-1] + branch1;//calculate the path cost given a 1
133       if(branch1 < mu_best[outputindex1][t-1]) { //check for overflow
134         branch1 = LARGE_NUMBER;
135       }
136
137       if ( branch0 <= branch1) {//Set the path cost in this point to the lowest of the
              two path costs
138         mu_best[i][t] = branch0;
139       } else {
140         mu_best[i][t] = branch1;
141       }
142     }
143   }
144
145   /* Backwards Run - The exact same as the forward run, just other indices */
146
147   for (t=BLOCK_SIZE+1; t>=0; t--) {
148     for (i=0; i< NSTATES; i++) {
149       // branch0
150       if (t >= BLOCK_SIZE) {
151         WR_SOVA1(sysref,65536);
152         WR_SOVA1(sysrec,-65536);
153       } else {
154         WR_SOVA1(sysrec,sys_internal[t]);
155         WR_SOVA1(sysref,-65536);
156       }
157
158       outpar1 = output[i+NSTATES][0];
159             outpar2 = output[i+NSTATES][1];
160
161       WR_SOVA1(par1rec,outpar1);
162       WR_SOVA1(par1ref,par1[t]);
163
164       WR_SOVA1(par2rec,outpar2);
165       WR_SOVA1(par2ref,par2[t]);
166
```

```
167
168         if (t >= BLOCK_SIZE) {
169           WR_SOVA1(log_in,0);
170         } else {
171           WR_SOVA1(log_in,La[t]);
172         }
173
174         branch0 = RD_SOVA1(branchres);
175
176         if (t > BLOCK_SIZE) {
177           WR_SOVA1(sysref,65536);
178           WR_SOVA1(sysrec,0);
179         } else {
180           WR_SOVA1(sysrec,sys_internal[t]);
181           WR_SOVA1(sysref,65536);
182         }
183
184         outpar1 = output[i][0];
185               outpar2 = output[i][1];
186
187         WR_SOVA1(par1rec,outpar1);
188         WR_SOVA1(par1ref,par1[t]);
189
190         WR_SOVA1(par2rec,outpar2);
191         WR_SOVA1(par2ref,par2[t]);
192
193         if (t >= BLOCK_SIZE) {
194           WR_SOVA1(log_in,0); // La_temp = 0;
195         } else {
196           WR_SOVA1(log_in,La[t]); //La_temp = La[t];
197         }
198
199         branch1 = RD_SOVA1(branchres);
200
201         branch0 = mu_best_back[transitions_back[i][0]][t+1] + branch0;
202         if(branch0 < mu_best_back[transitions_back[i][0]][t+1]) {
203           branch0 = LARGE_NUMBER;
204         }
205         branch1 = mu_best_back[transitions_back[i][1]][t+1] + branch1;
206         if(branch1 < mu_best_back[transitions_back[i][1]][t+1]) {
207           branch1 = LARGE_NUMBER;
208         }
209
210         if ( branch0 <= branch1) {
211           mu_best_back[i][t] = branch0;
212         } else {
213           mu_best_back[i][t] = branch1;
214         }
215       }
216     }
217
218     /* Calculation of soft-output and extrinsic information */
219
220     for (t=0;t<BLOCK_SIZE;t++) {
221       for (i=0; i < NSTATES; i++) {
222         //use the hardware accelerator to calculate the branch0
223
224         WR_SOVA1(sysrec,sys_internal[t]);
225         WR_SOVA1(sysref,-65536);
226
227         outputindex0 = transitions_back[i][0]; //Maybe transitions_back[i][0]
228         outpar1 = output[outputindex0+NSTATES][0];
229               outpar2 = output[outputindex0+NSTATES][1];
230
231         WR_SOVA1(par1rec,outpar1);
232         WR_SOVA1(par1ref,par1[t]);
233
234         WR_SOVA1(par2rec,outpar2);
235         WR_SOVA1(par2ref,par2[t]);
236
```

```
237         if (t-1 > BLOCK_SIZE) {
238           WR_SOVA1(log_in,0);
239         } else {
240           WR_SOVA1(log_in,La[t]);
241         }
242
243         branch0 = RD_SOVA1(branchres);
244
245         //Calculate the full cost of a path through this point and with a 0 in this branch
246         temp_cost = mu_best[i][t] + mu_best_back[outputindex0][t+1] + branch0;
247         if(((t <= 1024) && (temp_cost < mu_best[i][t])) || ((t > 1024) && (temp_cost <
               mu_best_back[outputindex0][t+1]))) { //check for overflow
248           temp_cost = LARGE_NUMBER;
249         }
250
251         if (temp_cost < mu0 || i == 0) {//if the calculated cost are smaller than the other
               costs calculated for a 0 branch
252           mu0 = temp_cost; //  Save the value
253         }
254
255         //use the hardware accelerator to calculate the branch1
256         WR_SOVA1(sysrec,sys_internal[t]);
257         WR_SOVA1(sysref,65536);
258
259         outputindex1 = transitions_back[i][1];
260         outpar1 = output[outputindex1][0];
261               outpar2 = output[outputindex1][1];
262
263         WR_SOVA1(par1rec,outpar1);
264         WR_SOVA1(par1ref,par1[t]);
265
266         WR_SOVA1(par2rec,outpar2);
267         WR_SOVA1(par2ref,par2[t]);
268
269         if (t-1 > BLOCK_SIZE) {
270           WR_SOVA1(log_in,0);
271         } else {
272           WR_SOVA1(log_in,La[t]);
273         }
274
275         branch1 = RD_SOVA1(branchres);
276
277         //Calculate the full cost of a path through this point and with a 1 in this branch
278         temp_cost = mu_best[i][t] + mu_best_back[outputindex1][t+1] + branch1;
279         if(((t <= 1024) && (temp_cost < mu_best[i][t])) || ((t > 1024) && (temp_cost <
               mu_best_back[outputindex1][t+1]))) { //check for overflow
280           temp_cost = LARGE_NUMBER;
281         }
282
283         if (temp_cost < mu1 || i==0) {//if the calculated cost are smaller than the other
               costs calculated for a 0 branch
284           mu1 = temp_cost; //  Save the value
285         }
286
287       }
288
289       LLR[t] = mu0 - mu1; //Calculate the soft output
290       La[t] = LLR[t] - 4*sys_internal[t] - La[t]; //calculate the extrinsic information
291     }
292   }
```

B.6 interleave.c

```
1  #include "turbo_coding.h"
2
3  void interleave(int LUT[], int vector[])
4  {
5    static int temp_arr[BLOCK_SIZE];
```

```
6    int k;
7
8    for(k=0; k<=BLOCK_SIZE-1; k++) {
9      temp_arr[k]=vector[k]; //Move the data to temp_arr
10   }
11   for(k=0; k<=BLOCK_SIZE-1; k++) {
12     vector[k]=temp_arr[LUT[k]]; //Move the data back from temp_arr but interleaved
13   }
14 }
```

B.7 deinterleave.c

```
1  #include "turbo_coding.h"
2
3  void deinterleave(int LUT[],int vector[])
4  {
5    static int temp_arr[BLOCK_SIZE];
6    int k;
7
8    for(k=0; k<=BLOCK_SIZE-1; k++) {
9      temp_arr[k]=vector[k]; //Move data to temp_arr
10   }
11   for(k=0; k<=BLOCK_SIZE-1; k++) {
12     vector[LUT[k]]=temp_arr[k]; //Move data back from temp_arr, but deinterleaved
13   }
14 }
```

B.8 math.c

```
1  #include "turbo_coding.h"
2  #define order_exp 12
3  #define order_ln 20
4  #define DELTAEXP 655
5  #define DELTALN1 1441792
6  #define DELTALN1_NONFACTOR 22
7  #define DELTALN2 64
8
9  #define CORR0 20530
10 #define FIRST_FACTOR 17625
11 #define SECOND_FACTOR 4295
12
13 int jacob_log(int x) {
14   int output, temp, first_term, second_term;
15
16   if (x < 0) { //calculate the max(0,x) term
17     output = 0;
18   }
19   else {
20     output = x;
21   }
22
23   if (x < 200004 && x > -200004) { //Check if the correction term should be used
24
25     if (x<0) { //abs(x) - 1
26       temp = -x - 65536;
27     } else {
28       temp = x - 65536;
29     }
30
31     if (temp > 32768 || temp < -32768) {
32       temp = temp >> 8;
33       first_term = temp * FIRST_FACTOR; //calculate the first order term
34       first_term = first_term >> 8;
35             second_term = temp*temp;
36       second_term = second_term * SECOND_FACTOR; //calculate the second order term
37       second_term = second_term >> 16;
```

```
38      } else {
39      //   printf("test\n");
40        first_term = temp * FIRST_FACTOR;
41        first_term = first_term >> 16; //calculate the first order term
42        second_term = temp*temp;
43        second_term = second_term >> 16;
44        second_term = second_term * SECOND_FACTOR;
45        second_term = second_term >> 16; //calculate the second order term
46      }
47      output = output + CORR0 - first_term + second_term; //calculate the output as the max
          (0,x) + correction
48    }
49    return output;
50  }
```

VHDL source code for hardware accelerator

C.1 bcc.vhd

```
1  library ieee, work;
2  use ieee.std_logic_1164.all;
3  use ieee.std_logic_unsigned.all;
4
5  entity bcc is -- Branch cost component
6      port (
7      signal clk,
8          reset_n,
9          read,
10         write,
11         chipselect : in std_logic;
12     signal readdata : out std_logic_vector(31 downto 0);
13     signal writedata : in std_logic_vector(31 downto 0);
14     signal address : in std_logic_vector(5 downto 0)
15   );
16 end bcc;
17
18 architecture behaviour of bcc is
19   -- For branch cost computation
20   signal sys_rec : std_logic_vector (31 downto 0); -- Systematic received
21   signal par1rec : std_logic_vector(31 downto 0); -- Parity 1 received
22   signal par2rec : std_logic_vector(31 downto 0); -- Parity 2 received
23
24   signal sys_ref : std_logic_vector(31 downto 0); -- Systematic reference
25   signal par1ref : std_logic_vector(31 downto 0); -- Parity 1 reference
26   signal par2ref : std_logic_vector(31 downto 0); -- Parity 2 reference
27
28   signal syscost, par1cost, par2cost: std_logic_vector(31 downto 0); -- Systematic and
       parity costs
29   signal branch_cost : std_logic_vector (31 downto 0); -- Final branch cost
30
31   -- For computation of Jacobian algorithm
32   signal x : std_logic_vector(31 downto 0); -- Input
33   signal jaclog: std_logic_vector(31 downto 0); -- Result
34   signal xtemp, logxmax, first_term, second_term: std_logic_vector(31 downto 0) :=
       std_logic_vector'("00000000000000000000000000000000"); -- Passing between processes
35   constant corr0: std_logic_vector(31 downto 0) := std_logic_vector'("
       00000000000000000101000000110010"); -- Correction factor of 20530
36   constant first_factor: std_logic_vector(15 downto 0) := std_logic_vector'("
```

```
      0100010011011001"); -- Factor of 17625
37   constant second_factor: std_logic_vector(15 downto 0) := std_logic_vector'("
      0001000011000111"); -- Factor of 4295 with point in 2^0
38
39   -- Misc. constants
40   constant ones: std_logic_vector(63 downto 0) := std_logic_vector'("
      1111111111111111111111111111111111111111111111111111111111111111");
41   constant zeros: std_logic_vector(63 downto 0) := std_logic_vector'("
      0000000000000000000000000000000000000000000000000000000000000000");
42
43   -- For the communication process
44   signal flag_rec : std_logic;
45   signal clk_cnt : std_logic_vector(9 downto 0);
46 begin
47
48 Avalon: process (clk)
49 variable sys_refee : std_logic_vector(31 downto 0);
50 begin
51   if clk'event and clk = '1' then
52    clk_cnt <= clk_cnt+1;
53     if reset_n = '0' then
54       readdata <= (others => '0');
55     else
56         if read = '1' then   -- if the BCC wants to write to Nios (avalon should read
            from BCC)
57           if address = 7 then
58               if clk_cnt > 2 and flag_rec = '1' then  -- a new branch cost has been
                  computed.
59                 readdata <= branch_cost; -- output this new branch cost.
60                 flag_rec <= '0';
61               else
62                 readdata <= "11111111111111111111111111111111"; -- return when nothing
                    has happened.
63               end if;
64           end if;
65         end if;
66
67          if write = '1' then  -- if write is asserted (avalon should write to the BCC)
68            -- The order in which inputs are fed from C is fixed
69
70             if address = 0 then      -- 0x00 (0)
71             sys_rec <= writedata;
72             elsif address = 1 then   -- 0x04 (4)
73             sys_ref <= writedata;
74             elsif address = 2 then   -- 0x08 (8)
75             par1rec <= writedata;
76             elsif address = 3 then   -- 0x0C (12)
77             par1ref <= writedata;
78             elsif address = 4 then   -- 0x10 (16)
79             par2rec <= writedata;
80             elsif address = 5 then   -- 0x14 (20)
81             par2ref <= writedata;
82             elsif address = 6 then   --0x18 (24)
83             x <= writedata;
84             flag_rec <= '1';
85             clk_cnt <= (others => '0');
86            else
87             null;
88
89            end if;
90
91          end if;
92     end if;
93   end if;
94 end process;
95
96 Psyscost: process (sys_rec, sys_ref) -- Compute cost of systematic bits
97   variable temp: std_logic_vector (62 downto 0); -- Temporary variable
98   variable rec, ref: std_logic_vector(31 downto 0); -- Converted par/outpar from two's
      complement
```

```
99    begin
100     if sys_rec(31) = '1' then -- Negative so convert from two's complement
101       rec := sys_rec(31) & (ones(30 downto 0) - sys_rec(30 downto 0) + '1');
102     else -- Positive so no conversion
103       rec := sys_rec;
104     end if;
105     if sys_ref(31) = '1' then -- Negative so convert from two's complement
106       ref := sys_ref(31) & (ones(30 downto 0) - sys_ref(30 downto 0) + '1');
107     else
108       ref := sys_ref;
109     end if;
110
111     -- temp = rec - ref, ignore sign as temp is squared later
112     if rec(31) = ref(31) then -- rec and ref both positive/negative
113       if rec(30 downto 0) > ref(30 downto 0) then
114         temp := rec(31) & zeros(30 downto 0) & (rec(30 downto 0) - ref(30 downto 0)); --
              Keep sign of either
115       else
116         temp := rec(31) & zeros(30 downto 0) & (ref(30 downto 0) - rec(30 downto 0));
117       end if;
118     else
119       temp := '0' & zeros(30 downto 0) & (rec(30 downto 0) + ref(30 downto 0)); -- would
            be either negative or positive but temp is squared next so doesn't matter!
120     end if;
121
122     temp := '0' & (temp(30 downto 0) * temp(30 downto 0)); -- Squaring
123     temp := temp(62) & zeros(15 downto 0) & temp(61 downto 16); -- Shifting down 16 bits,
          intro zeros, keep sign
124
125     -- Prepare for 32 bit representation
126     --   - check overflow (max 2^31-1)
127     --   - move sign to place 31
128     if temp(61 downto 31) > 0 then
129       temp := zeros(30 downto 0) & temp(62) & ones(30 downto 0);  -- set to 2^31-1, move
            sign, insert zeros
130     else
131       temp := zeros(30 downto 0) & temp(62) & temp(30 downto 0);  -- keep absolute value,
            move sign, insert zeros
132     end if;
133
134     syscost <= temp(31 downto 0);
135   end process;
136
137 Ppar1cost:  process (par1rec, par1ref) -- Compute cost of parity 1
138   variable temp: std_logic_vector (62 downto 0); -- Temporary variable
139   variable rec, ref: std_logic_vector(31 downto 0); -- Converted par/outpar from two's
        complement
140   begin
141     if par1rec(31) = '1' then -- Negative so convert from two's complement
142       rec := par1rec(31) & (ones(30 downto 0) - par1rec(30 downto 0) + '1');
143     else -- Positive so no conversion
144       rec := par1rec;
145     end if;
146     if par1ref(31) = '1' then -- Negative so convert from two's complement
147       ref := par1ref(31) & (ones(30 downto 0) - par1ref(30 downto 0) + '1');
148     else
149       ref := par1ref;
150     end if;
151
152     -- temp = rec - ref, ignore sign as temp is squared later
153     if rec(31) = ref(31) then -- rec and ref both positive/negative
154       if rec(30 downto 0) > ref(30 downto 0) then
155         temp := rec(31) & zeros(30 downto 0) & (rec(30 downto 0) - ref(30 downto 0)); --
              Keep sign of either
156       else
157         temp := rec(31) & zeros(30 downto 0) & (ref(30 downto 0) - rec(30 downto 0));
158       end if;
159     else
160       temp := '0' & zeros(30 downto 0) & (rec(30 downto 0) + ref(30 downto 0)); -- would
            be either negative or positive but temp is squared next so doesn't matter!
```

```
161       end if;
162
163       temp := '0' & (temp(30 downto 0) * temp(30 downto 0)); -- Squaring
164       temp := temp(62) & zeros(15 downto 0) & temp(61 downto 16); -- Shifting down 16 bits,
              intro zeros, keep sign
165
166       -- Prepare for 32 bit representation
167       --   - check overflow (max 2^31-1)
168       --   - move sign to place 31
169       if temp(61 downto 31) > 0 then
170         temp := zeros(30 downto 0) & temp(62) & ones(30 downto 0);  -- set to 2^31-1, move
                sign, insert zeros
171       else
172         temp := zeros(30 downto 0) & temp(62) & temp(30 downto 0);  -- keep absolute value,
                move sign, insert zeros
173       end if;
174
175       par1cost <= temp(31 downto 0);
176     end process;
177
178   Ppar2cost:  process (par2rec, par2ref) -- Compute cost of parity 2
179     variable temp: std_logic_vector (62 downto 0); -- Temporary variable
180     variable rec, ref: std_logic_vector(31 downto 0); -- Converted par/outpar from two's
          complement
181     begin
182       if par2rec(31) = '1' then -- Negative so convert from two's complement
183         rec := par2rec(31) & (ones(30 downto 0) - par2rec(30 downto 0) + '1');
184       else -- Positive so no conversion
185         rec := par2rec;
186       end if;
187       if par2ref(31) = '1' then -- Negative so convert from two's complement
188         ref := par2ref(31) & (ones(30 downto 0) - par2ref(30 downto 0) + '1');
189       else
190         ref := par2ref;
191       end if;
192
193       -- temp = rec - ref, ignore sign as temp is squared later
194       if rec(31) = ref(31) then -- rec and ref both positive/negative
195         if rec(30 downto 0) > ref(30 downto 0) then
196           temp := rec(31) & zeros(30 downto 0) & (rec(30 downto 0) - ref(30 downto 0)); --
                Keep sign of either
197         else
198           temp := rec(31) & zeros(30 downto 0) & (ref(30 downto 0) - rec(30 downto 0));
199         end if;
200       else
201         temp := '0' & zeros(30 downto 0) & (rec(30 downto 0) + ref(30 downto 0)); -- would
              be either negative or positive but temp is squared next so doesn't matter!
202       end if;
203
204       temp := '0' & (temp(30 downto 0) * temp(30 downto 0)); -- Squaring
205       temp := temp(62) & zeros(15 downto 0) & temp(61 downto 16); -- Shifting down 16 bits,
              intro zeros, keep sign
206
207       -- Prepare for 32 bit representation
208       --   - check overflow (max 2^31-1)
209       --   - move sign to place 31
210       if temp(61 downto 31) > 0 then
211         temp := zeros(30 downto 0) & temp(62) & ones(30 downto 0);  -- set to 2^31-1, move
                sign, insert zeros
212       else
213         temp := zeros(30 downto 0) & temp(62) & temp(30 downto 0);  -- keep absolute value,
                move sign, insert zeros
214       end if;
215
216       par2cost <= temp(31 downto 0);
217     end process;
218
219   Pbranch_cost: process (syscost,par1cost,par2cost,jaclog) -- Compute branch cost (x &
        sys_ref not in sens list as they are ready before costs & jaclog)
220       variable bc, logprob: std_logic_vector(31 downto 0);  -- Variable for adding the
```

```
        terms that make up the branch cost
221     variable xvar: std_logic_vector (31 downto 0);
222   begin
223     -- ADD COSTS TO BRANCH
224     bc := syscost; -- Start by setting bc equal to systematic cost
225
226     -- ADD par1cost
227     if bc(31)= par1cost(31) then -- Same sign then add
228       bc := bc(31) & (bc(30 downto 0) + par1cost(30 downto 0));
229     else -- Different sign then subtract
230       if bc(30 downto 0) > par1cost(30 downto 0)  then
231         bc := bc(31) & (bc(30 downto 0) - par1cost(30 downto 0));
232       else -- subtract reversed and flip sign
233         bc := ('1' nand bc(31)) & (par1cost(30 downto 0) - bc(30 downto 0));
234       end if;
235     end if;
236
237     -- ADD par2cost
238     if bc(31)= par2cost(31) then -- Same sign then add
239       bc := bc(31) & (bc(30 downto 0) + par2cost(30 downto 0));
240     else -- Different sign then subtract
241       if bc(30 downto 0) > par2cost(30 downto 0)  then
242         bc := bc(31) & (bc(30 downto 0) - par2cost(30 downto 0));
243       else -- subtract reversed and flip sign
244         bc := ('1' nand bc(31)) & (par2cost(30 downto 0) - bc(30 downto 0));
245       end if;
246     end if;
247
248     -- First turn jaclog into logprob depending on branch 0 or 1
249     -- requires computing x-jaclog so x should be converted from two's complement
250     if x(31) = '1' then -- Converting from two's complement
251       xvar := '1' & ( ones(30 downto 0) - x(30 downto 0) + '1' );
252     else
253       xvar := x; -- No conversion
254     end if;
255
256     if sys_ref(31) = '1' then -- branch0 (logprob = -jaclog)
257       logprob := ('1' nand jaclog(31)) & jaclog(30 downto 0); -- Multiply with -1 ie.
            flip sign
258     else -- branch1 (logprob = x-jaclog)
259       if xvar(31)=jaclog(31) then -- if same sign
260         if xvar(30 downto 0) > jaclog(30 downto 0) then
261           logprob := xvar(31) & (xvar(30 downto 0) - jaclog(30 downto 0));
262         else
263           logprob := ('1' nand xvar(31)) & (jaclog(30 downto 0)-xvar(30 downto 0));
264         end if;
265       else -- different sign
266           logprob := xvar(31) & (xvar(30 downto 0) + jaclog(30 downto 0));
267       end if;
268     end if;
269
270     -- SUBTRACT jacob_log
271     if bc(31) /= logprob(31) then  -- DIFFERENT sign then add and keep bc sign as a-(-b
          )=a+b or (-a)-b=-(a+b)
272       bc := bc(31) & (bc(30 downto 0) + logprob(30 downto 0));
273     else -- Different sign then subtract
274       if bc(30 downto 0) > logprob(30 downto 0) then
275         bc := bc(31) & (bc(30 downto 0) - logprob(30 downto 0));
276       else -- subtract reversed and flip sign
277         bc := ('1' nand bc(31)) & (logprob(30 downto 0) - bc(30 downto 0));
278       end if;
279     end if;
280
281     if bc(31) = '1' then -- Converting TO two's complement for negatives
282       bc := '1' & ( ones(30 downto 0) - bc(30 downto 0) + '1' );
283     end if;
284
285     branch_cost <= bc; -- Return accumulated branch cost
286   end process;
287
```

```
288 Pjaclog_start:  process (x) -- Start computation of Jacobian log. of x
289     variable xvar, logxvar, temp: std_logic_vector (31 downto 0);
290   begin
291     if x(31) = '1' then -- Converting from two's complement
292       xvar := '1' & ( ones(30 downto 0) - x(30 downto 0) + '1' );
293     else
294       xvar := x; -- No conversion
295     end if;
296
297     if xvar(30 downto 0) < 200004 then -- Check if Jacobian log. of x needs correction
298       -- Compute temp=|x|-65536 ie. subtract 1
299       if xvar(30 downto 0) >= 65536 then
300         temp := '0' & (xvar(30 downto 0) - 65536); -- Positive, subtract 2^16
301       else
302         temp := '1' & (65536 - xvar(30 downto 0)); -- Negative, add 2^16
303       end if;
304     else
305       temp := '1' & zeros(30 downto 0); -- Value -0 is used to signal 'no correction'
306     end if;
307     xtemp <= temp; -- Feed xtemp for computing 1st and 2nd term
308   end process;
309
310 P_1st_term: process (xtemp) -- Compute 1st term for correction of Jacobian log.
311     variable temp: std_logic_vector(62 downto 0) := zeros(62 downto 0);
312   begin
313     if xtemp = ('1' & zeros(30 downto 0)) then -- If 'no correction' then pass signal
314       temp := zeros(30 downto 0) & '1' & zeros(30 downto 0);
315     else -- Otherwise multiply with first_factor
316       temp := xtemp(31) & zeros(14 downto 0) & (xtemp(30 downto 0) * first_factor(15
            downto 0)); -- keep sign of xtemp, multiply with first factor which is positive
317       temp := zeros(30 downto 0) & temp(62) & temp(46 downto 16); -- Prepare for 32bit
            representation, keep sign, shift down 16 bits, no overflow possible
318     end if;
319     first_term <= temp(31 downto 0); -- Return first_term in 32 bits
320   end process;
321
322 P_2nd_term: process (xtemp) -- Compute 2nd term for correction of Jacobian log.
323     variable temp: std_logic_vector(62 downto 0) := zeros(62 downto 0);
324   begin
325     if xtemp = ('1' & zeros(30 downto 0)) then -- If 'no correction' then pass signal
326       temp := zeros(30 downto 0) & '1' & zeros(30 downto 0);
327     else -- Otherwise square and mul with second_factor
328       temp := '0' & xtemp(30 downto 0) * xtemp(30 downto 0); -- Squaring therefore
            positive sign
329       temp := temp(62) & zeros(15 downto 0) & temp(61 downto 16); -- Shift down 16 bits,
            introduce zeros, keep sign
330
331       temp := temp(62) & (temp(45 downto 0) * second_factor(15 downto 0)); --
            second_factor>0 so keep sign
332       temp := temp(62) & zeros(15 downto 0) & temp(61 downto 16); -- Shift down 16 bits,
            introduce zeros, keep sign
333
334       if temp(45 downto 31)>0 then -- Overflow check (prepare for 32 bit representation)
335         temp := zeros(30 downto 0) & temp(62) & ones(30 downto 0); -- set to 2^31-1, keep
              +move sign
336       else
337         temp := zeros(30 downto 0) & temp(62) & temp(30 downto 0); -- keep absolute value
              , keep+move sign
338       end if;
339     end if;
340     second_term <= temp(31 downto 0); -- Return second_term
341   end process;
342
343 Pjaclog_end:  process (first_term, second_term) -- Return result of Jacobian log. of x
344     variable temp: std_logic_vector(31 downto 0); -- With space for overflow from
          additions
345   begin
346     -- Converting x from two's complement
347     if x(31) = '1' then
348       temp := '1' & (ones(30 downto 0) - x(30 downto 0) + '1');
```

```
349       else
350         temp := x; -- No conversion
351       end if;
352
353       -- Initial rough estimation of Jacobian algorithm
354       if temp(31) = '1' then -- temp=0 below 0, temp=x above 0
355         temp := zeros(31 downto 0);
356       end if;
357
358     if (first_term /= '1' & zeros(30 downto 0)) and second_term /= ('1' & zeros(30 downto
          0)) then -- If 1st and 2nd term are not 'no corr' signals then do correction
359       -- Addition of corr0 (with overflow control)
360       if ones(30 downto 0)-temp(30 downto 0) < corr0(30 downto 0) then -- If overflow will
            happen
361         temp := temp(31) & ones(30 downto 0); -- set to max 2^31-1
362       else -- Add normally
363         temp := temp(31) & (temp(30 downto 0) + corr0(30 downto 0)); -- As they are both
              always positive (or zero)
364       end if;
365
366       -- Subtraction of 1st term (with overflow control) (term can be +/-)
367       if first_term(31)='1' then -- If sign is negative absolute value should be added
368         -- Overflow control on addition of absolute value of first_term
369         if ones(30 downto 0)-temp(30 downto 0) < first_term(30 downto 0) then -- If
              overflow will happen
370           temp := temp(31) & ones(30 downto 0); -- set to max 2^31-1
371         else -- Add normally
372           temp := temp(31) & (temp(30 downto 0) + first_term(30 downto 0)); -- absolute
                value of 1st term is positive
373         end if;
374       else  -- Subtract when sign is positive
375         if temp(30 downto 0) > first_term(30 downto 0) then -- just subtract
376           temp := temp(31) & (temp(30 downto 0) - first_term(30 downto 0));
377         else -- change sign
378           temp := '1' & (first_term(30 downto 0) - temp(30 downto 0));
379         end if;
380       end if;
381
382       -- Addition of 2nd term (with overflow control) (term always +)
383       if temp(31)='0' then
384         if ones(30 downto 0)-temp(30 downto 0) < second_term(30 downto 0) then -- If
              overflow will happen
385           temp := temp(31) & ones(30 downto 0); -- set to max 2^31-1, still positive
386         else -- Add normally
387           temp := temp(31) & (temp(30 downto 0) + second_term(30 downto 0)); -- Still
                positive sign
388         end if;
389       else -- if temp is negative then subtract 2nd term from absolute value
390         if temp(30 downto 0) > second_term(30 downto 0) then -- just subtract
391           temp := temp(31) & (temp(30 downto 0) - second_term(30 downto 0)); -- Still
                negative
392         else -- change sign
393           temp := '0' & (second_term(30 downto 0) - temp(30 downto 0));
394         end if;
395       end if;
396     end if; -- Else do not correct
397
398     jaclog <= temp; -- Return Jacobian logarithm (corrected or uncorrected)
399     end process;
400
401   end behaviour;
```

Bibliography

[Altera Corp., 2008] Altera Corp. (2008). Profiling Nios II systems.

[Altera Corp., 2009] Altera Corp. (2009). Nios II processor reference handbook.

[Bahl et al., 1974] Bahl, L., Cocke, J., Jelinek, F., and Raviv, J. (1974). Optimal decoding of linear codes for minimizing symbol error rate. *Information Theory, IEEE Transactions on*, 20(2):284–287.

[Bendat and Piersol, 1986] Bendat, J. S. and Piersol, A. G. (1986). *Random Data: Analysis and measurement procedures*. John Wiley & Sons, 2. edition.

[Carver and Tai, 2006] Carver, R. H. and Tai, K.-C. (2006). *Modern Multithreading*. Wiley-Interscience, 1. edition.

[Chaikalis et al., 2002] Chaikalis, C., Noras, J. M., and Riera-Palou, F. (2002). Improving the reconfigurable SOVA/log-MAP Turbo decoder for 3gpp. *CSNDSP 2002*, pages 105–108.

[Divsalar and Pollara, 1995] Divsalar, D. and Pollara, F. (1995). Turbo codes for pcs applications. *Communications, 1995. ICC '95 Seattle, 'Gateway to Globalization', 1995 IEEE International Conference on*, 1:54–59 vol.1.

[Dolinar and Divsalar, 1995] Dolinar, S. and Divsalar, D. (1995). Weight distributions for turbo codes using random and nonrandom permutations. *TDA Progress Report*, 42-122:56–65.

[Fan Mo, 1999] Fan Mo, Kwatra, S. J. K. (1999). Analysis of puncturing pattern for high rate turbo codes. *IEEE*, pages 547 – 550 vol.1.

[Fort et al., 2006] Fort, B., Capalija, D., Vranesic, Z., and Brown, S. (2006). A multithreaded soft processor for sopc area reduction. pages 131–142.

[Gibbons and Chakraborti, 2003] Gibbons, J. D. and Chakraborti, S. (2003). *Nonparametric Statistical Inference*. Marcel Dekker, Inc., 4. edition.

[Gries, 2004] Gries, M. (2004). Methods for evaluating and covering the design space during early design development. *Integr. VLSI J.*, 38(2):131–183.

[Hallinan, 2006] Hallinan, C. (2006). *Embedded Linux Primer: A Practical, Real-World Approach*. Prentice Hall.

[Haykin, 2001] Haykin, S. (2001). *Communication Systems*. John Wiley & Sons, 4. edition.

[Integrated Silicon Solution, 2006] Integrated Silicon Solution, I. I. (2006). IS61LV25616AL datasheet.

[Jin et al., 2006] Jin, Y., Zhang, F., and ling Wu, W. (2006). Reduced-complexity turbo equalization for turbo coded mimo/ofdm systems. *The Journal of China Universities of Posts and Telecommunications*, 13(1):93 – 98.

[Labrecque and Steffan, 2007] Labrecque, M. and Steffan, J. (2007). Improving pipelined soft processors with multithreading. pages 210–215.

[Langton, 2006] Langton, C. (2006). Turbo coding and MAP decoding.

[Liang et al., 2004] Liang, J., Tessier, R., and Goeckel, D. (2004). A dynamically-reconfigurable, power-efficient turbo decoder.

[Loudon, 1999] Loudon, K. (1999). *Mastering Algorithms with C*. O'Reilly.

[M. R. Soleymani, 2002] M. R. Soleymani, Y. G. U. V. (2002). *Turbo coding for satellite and wireless communications*. Kluwer Academic Publishers.

[Markowsky, 1992] Markowsky, G. (1992). Misconceptions about the golden ratio. *College Mathematics Journal*, 23(1):2–19.

[Perez et al., 1996] Perez, L., Seghers, J., and Costello, D.J., J. (1996). A distance spectrum interpretation of turbo codes. *Information Theory, IEEE Transactions on*, 42(6):1698–1709.

[Proakis and Salehi, 2002] Proakis, J. G. and Salehi, M. (2002). *Communication Systems Engineering*. Pearson Education, Inc., 2. edition.

[PSC, 2005] PSC, P. (2005). A2V64S40CTP 64 MB SDRAM Specification.

[Ratzer, 2001] Ratzer, E. A. (2001). Convolutional code performance as a function of decoding delay. *6^{th} International Symposium on Communication Theory and Applications*.

[Ross, 2004] Ross, S. M. (2004). *Introduction to Probability and Statistics for Engineers and Scientists*. Elsevier Inc., 3. edition.

[Rushton, 1998] Rushton, A. (1998). *VHDL for Logic Synthesis*. John Wiley & Sons.

[S. Crozier, 1999] S. Crozier, J. Lodge, P. G. e. A. H. (1999). Performance of turbo codes with relative prime and golden interleaving strategies. *Proceedings of the 6th International Mobile Satellite Conference (IMSC '99)*, pages 268–275.

[Sanchez et al., 2000] Sanchez, M., Shanmugan, K., De Haro, L., and Calvo, M. (2000). Burst error correction capabilities of turbo codes in mobile environments. *3G Mobile Communication Technologies, 2000. First International Conference on (Conf. Publ. No. 471)*, pages 176–180.

[Shannon, 1948] Shannon, C. E. (1948). A mathematical theory of communication. *Bell System Technical Journal*, 27:50–64.

[Sklar, 2002] Sklar, B. (2002). Fundamentals of turbo codes.

[Sripimanwat, 2005] Sripimanwat, K., editor (2005). *Turbo Code Applications - a journey from a paper to realization*. Springer.

[Sweeney, 2002] Sweeney, P. (2002). *Error Control Coding - From Theory to Practice*. John Wiley & Sons, 1. edition.

[Takeshita and Costello, 1998] Takeshita, O. and Costello, D.J., J. (1998). New classes of algebraic interleavers for turbo-codes. *Information Theory, 1998. Proceedings. 1998 IEEE International Symposium on*, pages 419–.

[Takeshita and Costello, 2000] Takeshita, O. and Costello, D.J., J. (2000). New deterministic interleaver designs for turbo codes. *Information Theory, IEEE Transactions on*, 46(6):1988–2006.

[Thitimajshima, 1995] Thitimajshima, P. (1995). Recursive systematic convolutional codes and application to parallel concatenation. volume 3, pages 2267–2272 vol.3.

[Vucetic and Yuan, 2000] Vucetic, B. and Yuan, J. (2000). *Turbo codes : Principles and Applications*. Kluwer Acedemic publishers, 1. edition.

[Wells, 1999] Wells, R. B. (1999). *Applied Coding and Information Theory for Engineers*. Prentice Hall, Thomas Kailath, 1. edition.

[Wicker and Kim, 2008] Wicker, S. B. and Kim, S. (2008). *Fundamentals of Code Graphs and Iterative Decoding*. Kluwer Academic Publishers, 1. edition.

[Woods et al., 2008] Woods, R., McAllister, J., Lightbody, G., and Yi, Y. (2008). *FPGA-based Implementation of Signal Processing Systems*. Wiley, 1. edition.

List of Abbreviations

ALU Arithmetic Logical Unit

ARQ Automatic Repeat Request

AWGN Additive White Gaussian Noise

BER Bit Error Rate

DFG Data Flow Graph

ECC Error Control Coding

FEC Forward Error Correction

FPGA Field-Programmable Gate Array

FU Functional Unit

FSM Finite State Machine

HDL Hardware Description Language

LE Logic Element

LLR Log Likelihood Ratio

LUT Look-up Table

MAC Multiply & Accumulate

NSC Non-Systematic Convolutional

PCCC Parallel Concatenated Convolutional Code

PDF Probabillity Density Function

PG Precedence Graph

VA Viterbi Algorithm

VHDL Very high speed integrated circuits Hardware Description Language

RSC Recursive Systematic Convolutional

SNR Signal-to-Noise Ratio

SISO Soft-Input Soft-Output

SOVA Soft Output Viterbi Algorithm

TC Turbo Coding

WE Write Enable

List of Figures

1.1 The A^3 framework model. Different algorithms are found to fulfil a single application, where after the algorithm of choice may be implemented on multiple types of architectures. The dashed arrows ensure compliance between the different modules so that the implementation in the end will fulfil the specification of the application. 4
1.2 The modified A^3 framework model. 5
1.3 The components of a typical communication system [Proakis and Salehi, 2002, p. 8]. 6
1.4 The Probability Density Function (pdf) (see 1.1) of Gaussian noise with variance 2. 7
1.5 A realisation of white Gaussian noise with variance 2 (plotted by use of the Matlab-function `randn`). 7
1.6 FFT of the realisation of white Gaussian noise depicted in Figure 1.5. 7
1.7 A simplified model of the communication system. 8
1.8 Convolutional encoder with $R = 1/2$ and $K = 3$ 10
1.9 State diagram of the system depicted in Figure 1.8. The states are determined and named by the content of the two delay elements while the transitions are marked with input/output to the states. 11
1.10 Trellis diagram of the system depicted in Figure 1.8. The transitions are represented by dashed lines for input sequence bits equalling 1 and solid lines for input sequence bits equalling 0. The transitions are marked with the corresponding output. 11
1.11 The RSC encoder with $R = 1/2$ and $K = 3$. 12
1.12 State diagram for the (1,5/7) RSC encoder depicted in Figure 1.11. The states and transitions are marked as on Figure 1.8. 12

2.1 Illustration of the current phase of the project with regards to the modified A^3 model. This is the phase for analysis of the algorithms for the different blocks which together form a turbo coder. 13
2.2 PCCC Turbo encoder with rate $R = 1/3$ 15
2.3 Recursive Convolutional encoder with an impulse as input. The systematic output is therefore also an impulse response. 16
2.4 Non-recursive Convolutional encoder with an impulse as input. The systematic output is therefore also an impulse response. 16

3.1 S-random method 1 illustrated as a flowchart 24
3.2 S-random method 2 illustrated as a flowchart 24
3.3 Scatter diagram of permutation by golden interleaver.

. 29

3.4 Scatter diagram of permutation by S-random interleaver (method 1), S=22, 4 iterations. 29
3.5 Scatter diagram of permutation by S-random interleaver (method 2), S=22, 1 iteration. 29
3.6 Scatter diagram of permutation by quadratic interleaver. 29
3.7 Scatter diagram of permutation by S-random interleaver (method 2), S=341, 1 iteration. 29

4.1 SISO decoder. [M. R. Soleymani, 2002, Fig. 2.1]. 34
4.2 Iterative decoder with two SISO decoders. [M. R. Soleymani, 2002, Fig. 2.2]. . . 34
4.3 Grid in which all the possible state-sequences in the encoder can be illustrated. The dashed line illustrates a possible path. 37
4.4 Trellis grid showing transmitted ones and zeros as solid and dashed lines respectively. 41
4.5 Trellis grid with path metrics along the branches. 42

5.1 The compliance test step of the modified A^3 model. 45
5.2 The structure of the MatLAB prototype of a Turbo Coding scheme 46
5.3 Simulation results from MatLAB. Each curve shows the result of a certain number of iterations as stated in the legend. 48

6.1 Illustration of the current phase of the project with regards to the modified A^3 model. This is the phase for implementation of the Turbo decoder and interleaver/deinterleaver on the soft-core processor. 52
6.2 Jacobian algorithm used to approximate the function $\ln(1 + e^x)$. Here it is shown both with and without the correction term. 54
6.3 A plot of the residual between the function $\ln(1 + e^x)$ and the approximation obtained by use of the Jacobian algorithm. Here it is shown both with and without the correction term. 54
6.4 Block diagram depicting the way the soft-core software implementation is built in C code. 56
6.5 Overview of some of the peripherals that can be integrated on the DE2-Board as IP cores using the Altera SOPC Builder software. 58
6.6 Overview of the communication between the host PC and the FPGA platform. Also shown is the internal communication in the platform, where the Nios II is the processor. 60
6.7 The system overview in Altera SOPC Builder, indicating address ranges for the different blocks, IRQs for the communication ports and overall interconnection. . 61

7.1 Illustration of the current phase of the project with regards to the modified A^3 model. This is the phase for implementation of the computationally intensive parts as hardware accelerators while leaving the other parts of the turbo decoder executing on the soft-core processor. 64
7.2 Illustration of solutions in the design space, where execution time and area are the cost metrics. The amount and positions of the solutions are merely examples, and does not represent ones specifically for this project. 68

7.3 The PG and DFG of the backwards and forward evaluation of the path cost. This process runs 8192 times for computation of all elements in `mu_best` and `mu_best_back`. 69
7.4 The PG and DFG of the output computation from the branch metrics. This algorithm runs 8192 times and will output the extrinsic values (La) and the soft-output (LLR) - one per four possible state transitions. 70
7.5 The PG and DFG of the `jacob_log` function which is called from the `sova` function in order to calculate the branch costs, `branch0` and `branch1`. More specifically, the logarithm function is called in order to calculate `logprob`. . . . 71
7.6 Structure of re-scheduling the sequential code to make the inherent parallelism more visible. 72
7.7 Structure of re-scheduling the sequential code to make the inherent parallelism visible. This is for the second part of the SOVA algorithm, where the extrinsic values, La, and the soft-outputs, LLR, are calculated 73
7.8 Structure of re-scheduling the sequential code for the `jacob_log` function to make the inherent parallelism more visible. 74
7.9 Structure of the loop-unfolding model for calculation of branch costs, best paths and outputs. Due to dependices from earlier iterations, the time for computing the extrinsic values La increases by one block of `branch0` and `branch1` for each parallel entity consisting of the computation of forward, backwards path cost and outputs. 75
7.10 The selected parts of the algorithms for the hardware implementation. 75
7.11 Overview of the functional processes in the hardware accelerator. Square boxes represent processes and arrows represent signals. 77
7.12 Overview of the communication signals and communication process in the hardware accelerator for interfacing to the Avalon bus. 78

8.1 Results for the execution time between the soft-core and the soft-core with hardware acceleration implementations. 82
8.2 Results for the BER achieved at different levels of SNRs for the prototype in MatLAB and the two implementations. The numbers next to the curves indicate the number of iterations for the specific curve. 83

List of Tables

1.1 Truth table for the NSC encoder illustrated in Figure 1.8. The states are numbered by the content of the two delay elements. 11

3.1 Results for the runs test for randomness of the various described interleaving methods. 30

6.1 Listing of the memory requirement for each of the C code functions which are implemented. This accounts only for the data storage, not the program by itself, and it does not account for the fact that variables may not require storage simultaneously. 56
6.2 Comparison of the three different Nios II processors implemented on the Cyclone II type. 59

7.1 Flat profile showing the time spent by selected functions, where some of these belong to the profiler libraries. 65
7.2 The summary report showing time spent by functions and number of times each of these are called. 66

8.1 Results of the first test, where execution time is presented with the number of decoding iterations. 83
8.2 Results of the second test, where the averaged BER is determined at 11 different SNRs, at three levels of iterations. The averaging is based on 20 repetitions for each SNR and for each number of iterations. 84