EINDHOVEN UNIVERSITY OF TECHNOLOGY

Eindhoven University of Technology

MASTER

Optimizations for the implementation of a turbo decoder

Maessen, F.A.M.

Award date:
2000

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

Optimizations for the implementation of a turbo decoder

Francien Maessen

November 2000

Supervisors:

Dr. Ir. F.M.J. Willems (Eindhoven University of Technology)
Prof. Dr. Ir. G. Brussaard (Eindhoven University of Technology)
Prof. Dr. Ir. L. van der Perre (IMEC)

Abstract

Turbo decoding techniques achieve near-optimal performance in terms of Bit Error Rate (BER) at low Signal to Noise Ratios (SNR's). Amongst these techniques, convolutional turbo codes and product turbo codes are nowadays the most common ones. They have already been selected as a standard for the Universal Mobile Telecommunication System (UMTS) [Lit. 4]. In the future, they will probably also be selected as a standard for Wireless Local Area Networks (WLAN).

For such high-speed wireless applications, throughput and energy consumption are crucial issues. By contrast, the hardware implementation of the turbo codes is rather slow and power consuming. Therefore, the implementation of turbo decoders needs to be optimized.

Since an analysis of the energy consumption and throughput in turbo decoders has indicated a bottleneck in memory accesses, a systematic data transfer and storage optimization methodology, developed at IMEC, has been applied. It reduces the energy consumption and latency; in addition, it allows reaching higher data rates. The whole methodology has been applied to the convolutional turbo decoder, while for the product turbo decoder only some global optimizations have been exploited.

Based on a high-level memory- and architectural model, estimations have been made for area, energy per bit, throughput and latency at a constant clock frequency of 77 MHz. For the convolutional turbo code, a 25-fold energy reduction per decoded bit has been achieved, while at the same time the speed is multiplied by 400 and the latency divided by 400. These results have been achieved at the cost of the logic- and memory area: the total area consumption is increased by a factor of 5. The results of the product turbo code show a gain in both energy- and area consumption: the energy consumption per decoded bit is decreased with a factor 5 and the area consumption is decreased with a factor 3. For the throughput and latency of the product turbo code, no estimations have been made in this report.

The energy consumption per decoded bit of the convolutional turbo code after optimization (0.04 µJ) has acceptable levels for an implementation at high data-rates. The product turbo code, however, still needs some optimizations before it can be implemented on a single chip, as the energy consumption per bit (0.6 µJ) is still too large for a high-speed implementation.

Acknowledgements

I am indebted to many people for their advice, assistance and contributions to my project on turbo coding.

First, I wish to thank the people at IMEC with whom I had many discussions and who gave me much inspiration. I especially would like to thank my supervisor at IMEC, Liesbet van der Perre, for her support on this project. She introduced me to "the world of scientists" by making me co-writer of some papers ([Lit. 22] and [Lit. 25]) and letting me submit a paper [Lit. 26] at the Symposium of Vehicular Technology and Communications. Furthermore, she gave me the opportunity to go to the 2nd International Symposium on Turbo Codes in Brest, which was a wonderful experience. My special thanks also to Marc Engels who gave me his support when Liesbet took maternity leave.

Furthermore, I would like to thank my supervisors from the Eindhoven University of Technology. First, I wish to thank Frans Willems, my direct supervisor, for his constant support and encouragement throughout the whole work on this project. He also advised me to work on the implementation of turbo decoders instead of the algorithm itself. Although I doubted his advice in the beginning, I am very glad that I took the decision to work on the implementation: it has been a very interesting project. In addition, I would like to thank my other supervisor from the Eindhoven University of Technology, Gert Brussaard, for his support. At times that I needed him, he was always ready to help.

Last, but not least, I would like to thank my boyfriend Arjan Mels for the many discussions and inspirations, for reviewing my report and for his constant support.

Francien Maessen

Table of contents

Abstract 2

Acknowledgements 3

Table of contents 4

1 Introduction 6

2 Fundamentals of convolutional turbo coding 7
- 2.1 The general transmission scheme 7
- 2.2 Encoding 8
- 2.3 Channel 8
- 2.4 Decoding 9
- 2.5 Convolutional codes 10

3 Basic implementation of the MAP-algorithm 15
- 3.1 Basic implementation 15
- 3.2 Identifying the bottlenecks 15

4 Optimizations for the convolutional turbo decoder 21
- 4.1 Data Transfer and Storage Exploration methodology 21
- 4.2 Global loop transformations 22
- 4.3 Memory hierarchy 25
- 4.4 Memory allocation 30
- 4.5 Data-flow transformations 34
- 4.6 Optimized convolutional decoder 36

5 Product turbo code 38
- 5.1 The algorithm 38
- 5.2 Optimizations for the product turbo decoder 40
- 5.3 Results 42

6 Conclusions 44

7 Future work 45

Acronyms 47

Literature 48

Appendix A Alcatel 0.35-μm CMOS technology 50
A.1. Memories 50
A.2. Registers 50
A.3. Arithmetic units 50
A.4. Length of the critical path 51

Appendix B Mathematical derivations in the MAP-algorithm 52
B.1. Derivation of the *LLR* 52
B.2. Logarithmic notation for *LLR* 53
B.3. Logarithmic notation for metric 54
B.4. Logarithmic notation for extrinsic information 56

Appendix C Implementation of calculations 57
C.1. Basic implementation metric calculation 57
C.2. Basic implementation input calculation 58
C.3. Basic implementation subtractive normalization 59
C.4. Basic implementation extrinsic output 60
C.5. Optimized implementation metric calculation and normalization 61

Appendix D Computations on the convolutional turbo code 62
D.1. Computations after loop transformations 62
D.2. Computations after introducing a memory hierarchy 63
D.3. Computations after memory allocation 63
D.4. Computations after data-flow transformations 64

Appendix E Output from ATOMIUM 66

Appendix F Derivation of LLR for product codes 67

Appendix G Sorting algorithms 68
G.1. Bubble sort 68
G.2. Shell's law 68

1 Introduction

Noise in a transmission channel is the limiting factor for the faithful replication of a transmitted signal. Shannon stated that it is possible, however, to establish error-free communication below a certain data-rate limit [Lit. 1]. This limit is called the Shannon capacity limit. Finding codes that operate near the Shannon capacity limit with a tolerable complexity is a challenge in information- and coding theory.

In [Lit. 2] a new class of codes, called turbo codes, was presented. The performance of these codes is close to the Shannon limit. A typical turbo code consists of two rather simple codes concatenated by an interleaver. For these simple codes, either convolutional- or product codes are commonly used. The main innovation of turbo decoding is to perform the decoding iteratively where soft output information of the first decoder is passed on to the input of the second decoder. Then the soft output of the second decoder is fed back to the first decoder and so on. This functionality reminded the authors of [Lit. 2] of a turbo engine and inspired them to this name.

The outstanding Bit Error Rate (BER) performance of this coding scheme created a large interest in turbo codes due to the wide range of possible applications. For example, a recent proposal for the Universal Mobile Telecommunication System (UMTS) for 3rd generation mobile communication includes a turbo-coding scheme [Lit. 4].

However, the implementation of a turbo decoder faces a number of challenges. UMTS applications and Wireless Local Area Networks (WLAN) for example demand a high-throughput decoder. This report focuses on the implementation of a turbo decoder that will be used for WLAN. Since this application needs a high-throughput decoder, the energy per decoded bit has also to be minimized in order to get reasonable power consumption.

With the purpose of reaching a higher throughput and lower energy consumption, we have applied a systematic data transfer and storage optimization technique, developed at the Inter-university Micro-Electronic Center (IMEC) [Lit. 5], to the turbo decoder. The whole methodology has been applied to the convolutional turbo code. The first global optimization steps have been investigated for the product turbo decoder in order to initiate a comparison with the convolutional turbo decoder.

This report is organized as follows: in Chapter 2, the fundamentals of coding and especially the turbo coding theory for convolutional turbo codes are explained; Chapter 3 describes the basic implementation of the convolutional turbo decoder and the bottlenecks of this implementation; Chapter 4 explains the optimizations applied to the convolutional turbo code; Chapter 5 describes the product turbo code and the optimizations applied to this code; finally, conclusions and recommendations can be found in Chapter 6 and Chapter 7.

2 Fundamentals of convolutional turbo coding

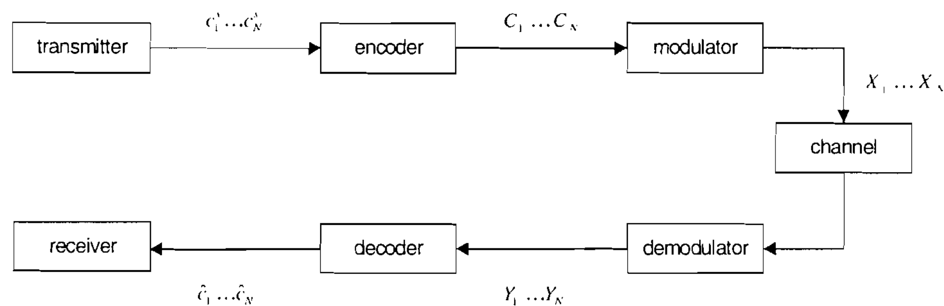
The goal of communication is to send messages from one point to another. Before sending, the messages are translated into physical properties, for example into electrical voltages. These electrical signals can be transmitted over a link, usually called the transmission channel. Unfortunately, the transmission channel adds noise to the transmitted signal. This noise leads to detection errors at the receiver side. The aim of the communication system is to reduce the number of transmission errors as much as possible.

In [Lit. 1], Shannon introduced the concept of channel capacity. He showed that it is possible to transmit information over a channel with an error probability approaching to zero for every rate less than the channel capacity. In [Lit. 2] a new class of codes with a performance close to the Shannon limit was presented. These codes are called turbo codes.

This chapter first describes the general model of a transmission scheme; subsequently, some elements of this scheme are explained for convolutional turbo codes in more detail.

2.1 The general transmission scheme

Figure 2.1 shows a block scheme of the transmission process. This process starts with generating binary information at the transmitter side that is split into so-called *frames*. These frames are fed into the encoder. The encoder adds *n-1*



• Figure 2.1: general transmission scheme[1]

redundant bits ($c_t^{p_1} \ldots c_t^{p_{n-1}}$) to each input bit ($c_t^s$) within a frame. The notation for the produced sequence of symbols generated by the encoder is given below.

$$C_1,\ldots,C_N\ ; \quad C_t = \begin{pmatrix} c_t^s \\ c_t^{p_1} \\ \vdots \\ c_t^{p_{n-1}} \end{pmatrix}; \quad c_t^{p_k} \in \{0,1\},\ t \in [1,N],\ k \in [1,n-1] \tag{2.1}$$

This sequence is the input to the modulator. The transformation of this sequence to a signal, which can be sent over the channel, can be done in various ways.

[1] This scheme is a simplified scheme and does not include for example compression

The transformation is for instance Binary Phase Shift Keying (BPSK). The sequence is thus transformed by using the transformation: $\{0,1\} \rightarrow \{-1,1\}$.

Hence, each symbol that is inserted into the channel can be calculated by

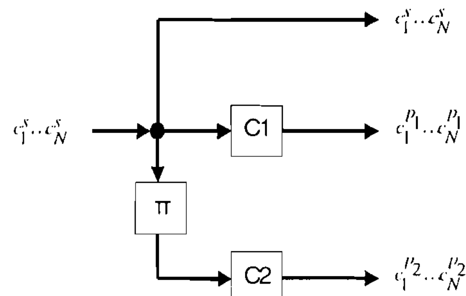
$$X_i = 2 \cdot C_i - 1 \tag{2.2}$$

The channel distorts the signal so that the decoder on the receiver side receives the sequence $Y_1 \dots Y_N$ instead of $C_1 \dots C_N$. The decoder provides an estimate $\hat{C}_1 \dots \hat{C}_N$ of the sent sequence.

In the following sections, the encoder, the channel and the decoder will be explained in more detail.

2.2 Encoding

The aim of an encoder is to map the incoming message to a code word. Figure 2.2 represents the encoding part of a turbo code: a parallel concatenation of two rather simple convolutional encoders (C1 and C2). The output symbol C_t of the turbo encoder is a combination of the original (*systematic*) sequence, one check bit from C1 and one check bit from C2 (the *parity* bits). The input to C1 is the original data sequence, while an interleaver π reorders the input to C2. Interleaving is done so that the inverse operation at the receiver side spreads neighboring errors. These spread erroneous symbols are less correlated for the decoder.



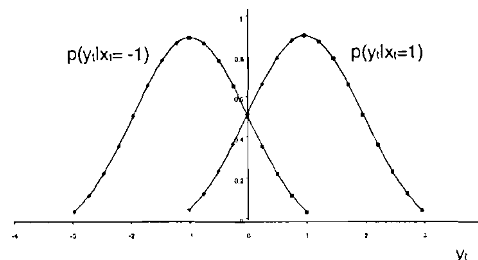
- Figure 2.2: turbo encoder

2.3 Channel

As explained before, time discrete signals with values *−1* and *+1* are sent over a channel. If *+1* has been transmitted whereas the decoder estimates *−1* due to the noise in the channel, a bit error occurs. The noise in a channel highly depends on the type of the channel. This section briefly describes the Additive White Gaussian Noise (AWGN) channel and the Rayleigh fading channel.

2.3.1 AWGN channel

The noise n of the AWGN channel is modeled by a zero-mean normally distributed random signal of variance $\sigma^2 = N_0/2$. N_0 denotes the one-sided spectral noise density. Let E_s be the average energy received for each symbol. Then E_s / N_0 is defined as the Signal to Noise Ratio (SNR).



- Figure 2.3: Gaussian model

The noise is additive. The received signal is thus: $y_t = x_t + n_t$. The AWGN channel is described by the following Power Distribution function (PDF) of the received signal:

$$p(y_t|x_t) = \frac{1}{\sqrt{2\pi\sigma^2}} \cdot \exp\left(-\frac{(y_t - x_t)^2}{2\sigma^2}\right) \tag{2.3}$$

The PDF model for the Gaussian channel is illustrated in Figure 2.3.

2.3.2 Rayleigh fading channel

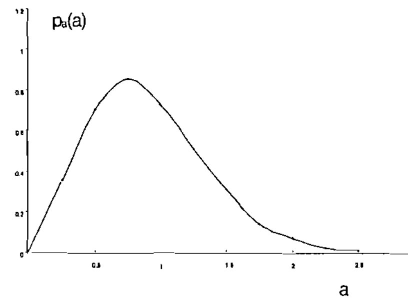
Channels that change their properties (for example due to multi-path propagation in mobile communications) during the transmission are called *time varying.* An exact model of these channels is very complex. That is why the model is simplified to the main properties of the channel, namely the value and the correlation of the received signal amplitude. A Rayleigh fading channel is in the study of communication systems an AWGN channel with varying signal amplitude a_t:

$$y_t = a_t \cdot x_t + n_t$$

The multiplicative deterioration a_t is Rayleigh distributed [Lit. 6] and given by the following formula:

$$p_a(a) = 2 \cdot ae^{-a^2} \qquad a > 0$$

The model for the Rayleigh distribution is represented in Figure 2.4.



• Figure 2.4: Rayleigh distribution

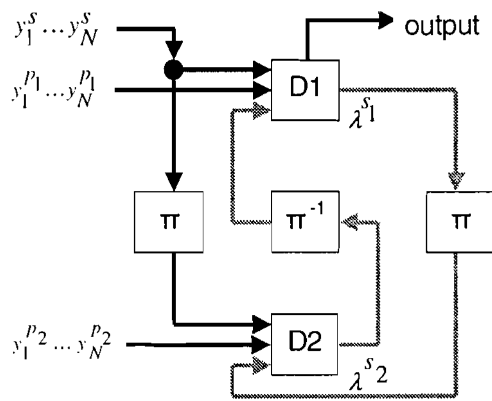
The Rayleigh channel is therefore described by the following PDF for the received signal:

$$p(y_t|a_t x_t) = \frac{1}{\sqrt{2\pi\sigma^2}} \cdot \exp\left(-\frac{(y_t - a_t x_t)^2}{2\sigma^2}\right) \tag{2.4}$$

2.4 Decoding

Turbo decoding is based on an iterative process, where two soft output decoders use each other's (interleaved) output as a-priori information.

The turbo decoding principle is outlined in Figure 2.5. Each encoder Ci of Figure 2.2 corresponds to a soft output decoder Di in Figure 2.5. The decoder operates on three input sequences: the original input and the convoluted sequence of the appropriate encoder, which have been transmitted over a channel and the a-priori information that is given by the previous decoding step. In the first decoding step, the a-priori information is 0.5 both *zero* and *one.* The decoding starts with D1. The output of this decoder lies in the range [-1,1]. The sign of the output gives the hard decision, while the absolute value gives information concerning the reliability of the hard decision. The output is split into *channel values, a-priori information*



• Figure 2.5: turbo decoder

and *extrinsic information*. The second decoder D2 uses the *interleaved extrinsic information* from D1 as a-priori information. The flow of the extrinsic information is symbolized with gray arrows in Figure 2.5. In the next iteration (a combination of D1 followed by D2 is called an iteration), D1 is executed again, now using the de-interleaved extrinsic information from D2 as a-priori information. This loop is continued until some stop criterion is met.

There is no fixed rule when to stop the iteration process although some criteria have been recommended in [Lit. 7] and [Lit. 8]. A higher number of iterations yields a better decoding performance but requires also a larger amount of time. Furthermore, the decoding gain of additional iterations decreases rapidly.

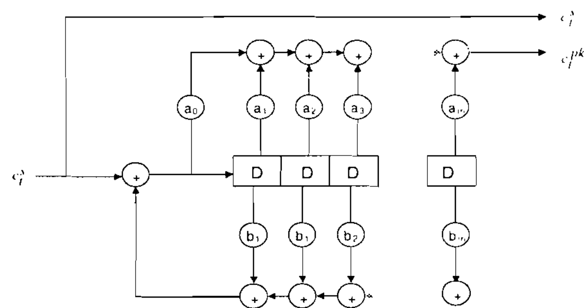
In the following section, the operation blocks of the encoder (C1, C2) and decoder (D1, D2) will be explained in more detail.

2.5 Convolutional codes

In the convolutional turbo coder, the convolutional code blocks are the most important modules. Therefore, the principle of the convolutional- encoding and decoding is given in this section.

2.5.1 Convolutional encoder

A convolutional encoder performs a discrete convolution using binary addition and multiplication: the input bits are sent through a shift register of known length *m,* while the output is a linear combination of the different register contents that are formed by using the parameters a_0, a_1, ... a_m (Figure 2.6). The *D* operator describes a delay of one register cell. When the code is recursive, another linear combination of the register contents is fed back to the input of the encoder. This feedback is described by the parameters b_1, b_2, ... b_m.



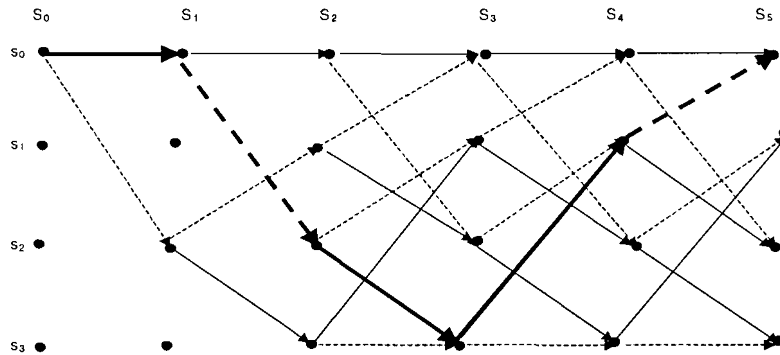
• Figure 2.6: systematic convolutional encoder

Since all operations within the convolutional encoder are linear, the convolution itself is also linear. The resulting convolution can be described by the following transfer function:

$$G(D) = \frac{a_0 + a_1 D + \ldots + a_m D^m}{1 + b_1 D + \ldots + b_m D^m} \qquad (2.5)$$

The code is called a *systematic* code if the input bit is also fed to the output without coding (like in Figure 2.6).

A sequence of transitions in a Finite State Machine (FSM) can describe the encoding process: the register content is represented by the current state s_i, denoted as S_t. Given the current state, the next input bit determines the next state. The output bit for a certain state-transition depends on three items: the current state, the input bit and the implementation of equation 2.5. The state transitions can be illustrated by a trellis graph showing the transitions between the different states.



• Figure 2.7: trellis of a four-state convolutional code

Figure 2.7 shows an example of the coding process by using a trellis. Solid lines correspond to transitions caused by the input of a *0*; dashed lines denote the injection of a *1*. Given a transfer function with a nominator $1 + D^2$ and a denominator $1 + D + D^2$, input {0,1,0,0,1} produces an output {0,1,1,0,1} of the encoder. The resulting path $\{S_0, S_1, S_2, S_3, S_4, S_5\} = \{s_0, s_0, s_2, s_3, s_1, s_0\}$ is highlighted in Figure 2.7. If the code is systematic, the following symbols will be transferred to the modulator:

$$\vec{c} = \left\{ \begin{pmatrix} 0 \\ 0 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \end{pmatrix} \right\}$$

By convention, the encoder starts in a known state: the all-zero state ($S_0 = s_0$). In certain cases, the encoder must end in a known state too. This can be achieved by forcing the encoder to a certain state by injecting *m* tail bits. The value of the tail bits depends on the current state of the FSM.

In this project, we assume a 8-state convolutional code with the following transfer function:

$$G(D) = \frac{1 + D + D^3}{1 + D^2 + D^3} \tag{2.6}$$

2.5.2 Convolutional decoder

The decoder module is the most complex block in the convolutional turbo decoder. The module can be implemented in several ways. In this report, the Maximum A-Posteriori (MAP) decoding algorithm introduced by Bahl et. al. [Lit. 9]

is used for decoding. The MAP-algorithm did not become very popular until recently because of its large latency and high energy consumption compared to for example the Viterbi algorithm [Lit. 3], which has a similar performance for low BER's. However, turbo codes have changed the popularity of the MAP-algorithm because they require soft output and the algorithms like the standard Viterbi algorithm do not provide this.

The MAP-algorithm has three inputs: a systematic input, an encoded input and a-priori information (see Figure 2.5). These inputs are used for calculating a forward- and a backward recursion. These calculations are done by reconstructing the trellis of the encoding process. An element within a recursion, called a *metric*, consists of a certain number of states that is defined by the trellis of the encoding process. A metric indicates the probability of occurrence for each state in the trellis. In order to calculate one output-bit ($\hat{c}_t^s$), one metric from the forward recursion and one metric from the backward recursion is needed.

The metrics in the forward- and backward recursion are introduced as α, respectively β in [Lit. 9]:

$$\alpha_t(S_t) = \sum_{S_{t-1}} \alpha_{t-1}(S_{t-1}) \cdot P_{metric-transition}(S_{t-1}, S_t) \tag{2.7}$$

$$\beta_t(S_t) = \sum_{S_{t+1}} \beta_{t+1}(S_{t+1}) \cdot P_{metric-transition}(S_t, S_{t+1}) \tag{2.8}$$

In order to compute these recursive values, a start value is needed. As said before, state *0* will be the first state and the last state. Since the metrics indicate the probability of the states, the initial value of the first state will be one, the others zero. Using the forward- and backward probabilities, the bit probabilities can be determined:

$$P_{a-posteriori}(c_t^s = 0) = \sum_{(s',s) \in B_0} \alpha_{t-1}(s') \cdot P_{metric-transition}(s', s) \cdot \beta_t(s) \tag{2.9}$$

$$P_{a-posteriori}(c_t^s = 1) = \sum_{(s',s) \in B_1} \alpha_{t-1}(s') \cdot P_{metric-transition}(s', s) \cdot \beta_t(s) \tag{2.10}$$

where B_i is the set of transitions S_{t-1} = s' → S_t = s such that $c_t^s = i$.

In the probabilities given above, the *metric-transition probability* (that is the probability that a code word follows any path via state *s* at time *t*) is used. This probability can be formulated as follows:

$$P_{metric-transition}(s', s) = P(S_t = s, Y_t | S_{t-1} = s')$$

$$= P(Y_t | S_{t-1} = s', S_t = s) \cdot P(S_t = s | S_{t-1} = s', c_t^s = i) \cdot P(c_t^s = i)$$

where

- $P(Y_t | S_{t-1} = s', S_t = s)$ must be deduced from the channel characteristics
- $P(S_t = s | S_{t-1} = s', c_t^s = i)$ is either *0* or *1* depending on whether the input of c_t^s leads from state $S_{t-1} = s'$ to $S_t = s$
- $P(c_t^s = i)$ is the a-priori probability of the transition $s' \rightarrow s$

If the second probability is *1*, the metric-transition probability can be written as:

$$P_{metric-transition} = P(Y_t|C_t) \cdot P(c_t^s(s',s) = i) \tag{2.11}$$

With the Rayleigh model for the channel described in Section 2.3.2 and substituting X as in equation 2.2, this results in:

$$P_{metric-transition} = e^{-\left(\frac{|Y_t - a\cdot(2C_t(s',s)-1)|^2}{2\sigma^2}\right)} \cdot P(c_t^s(s',s) = i) \tag{2.12}$$

In order to generate one soft output decision, it is beneficial to formulate the bit probabilities as a *Log-Likelihood Ratio* (LLR) for two reasons: the received values can be easily converted to a LLR and mathematical simplifications are possible for the MAP decoding algorithm when using LLR. The mathematical derivation for the LLR is given in Appendix B. The outcome of this derivation is given below:

$$LLR\,(c_t^s) = \log \frac{P_{a-posteriori}(c_t^s = 1)}{P_{a-posteriori}(c_t^s = 0)}$$

$$= \frac{2a}{\sigma^2} \cdot y_t^s + L_{a-priori}(c_t^s) + L_{extrinsic}(c_t^s) \tag{2.13}$$

The first term in 2.13 is the *systematic channel value* and the second term represents the *a-priori information* that was inserted into the algorithm. The third term is the *extrinsic information.* This extrinsic information is used as a-priori information for the next decoding step (see Figure 2.5).

The derivation of the soft output shows a lot of multiplication and logarithmic functions. Due to this, a hardware implementation will be very expensive. Therefore, a transfer of the algorithm to the logarithmic domain is used. Thus, multiplication becomes addition.

A problem arises in the transformation of additions to the log-domain. This operation will be depicted as $\circledcirc$. Let $\tilde{\varphi}_1$ and $\tilde{\varphi}_2$ be two variables in the log domain[1]. An addition transformed to the log domain will look like:

$$\begin{aligned}
\tilde{\varphi}_1 \circledcirc \tilde{\varphi}_2 &= \ln\left(e^{\tilde{\varphi}_1} + e^{\tilde{\varphi}_2}\right) \\
&= \max(\tilde{\varphi}_1, \tilde{\varphi}_2) + \ln\left(1 + e^{-|\tilde{\varphi}_1 - \tilde{\varphi}_2|}\right) \\
&= \max(\tilde{\varphi}_1, \tilde{\varphi}_2) + f_c(|\tilde{\varphi}_1 - \tilde{\varphi}_2|) \\
&= \max{}^*(\tilde{\varphi}_1, \tilde{\varphi}_2)
\end{aligned} \tag{2.14}$$

This means that an operation $\circledcirc$ can boil down to the maximum operation with a correction function f_c *(max*)*. The max* function is described in [Lit. 12]. However, using fixed point values, the complexity of the implementation of the max* function can be significantly reduced. This implementation will be explained in the next chapter.

In Appendix B, the logarithmic derivations for the *LLR,* the metric values and the extrinsic information are given.

---

[1] the ~ indicates a variable in the log-domain

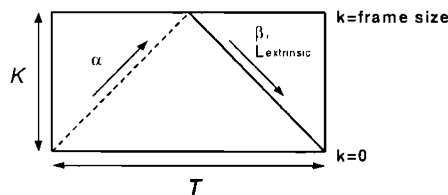
As mentioned before, the MAP-algorithm is the most important module within the turbo decoder. At the same time, it is also the most complex block to implement in hardware. In the next chapter, we will look at the basic implementation of MAP-algorithm in hardware.

3 Basic implementation of the MAP-algorithm

This chapter shows the basic implementation of the main building block of the convolutional turbo decoder: the MAP-decoder. Given the basic implementation, we can identify the bottlenecks of the decoder in terms of energy, area, throughput and latency for applications like WLAN.

3.1 Basic implementation

As can be concluded from the algorithm description, the decoder produces extrinsic information using forward- and backward probabilities. These probabilities are based on the entire frame: for the production of extrinsic information concerning the t-th bit, α_{t-1} and β_t have to be known. The most straightforward implementation calculates the α's over the entire frame and then the β's and extrinsic information in the backward direction, as shown in Figure 3.1. The x-axis indicates the time, whereas the y-axis indicates the recursion- or output calculation of a certain bit. K is the frame-size and T is the decoding latency for one half-iteration. The frame-size is assumed 400 in this report.



• Figure 3.1: basic implementation of the MAP-decoder

3.2 Identifying the bottlenecks

In order to identify the bottlenecks in the basic implementation of the MAP-algorithm, we have to make estimations for energy- and area consumption, as well as for throughput and latency. All estimations for the basic implementation have been based on the 0.35-µm CMOS process of Alcatel Microelectronics.

3.2.1 Estimations for energy and area

The costs in terms of energy and area depend on three aspects: control, data-path calculations and memory.

The control can be implemented as a finite state machine (FSM). This FSM can be easily implemented by using counters. Since the costs in terms of energy and area for a counter are rather low (see Appendix A), we did not take into account the energy- and area consumption for the control.

In order to be able to estimate the energy- and area consumption in the data-path and the memory, it should be identified:

- how the metric- and extrinsic calculations can be implemented in hardware;
- what the number of arithmetic operations, the number of memories and the number of accesses to these memories are;
- what the costs of a single arithmetic operation and a single memory (access) in terms of energy and area are;

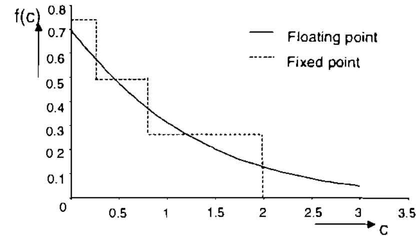
These three issues are investigated in the following sections.

Implementation issues for the calculations

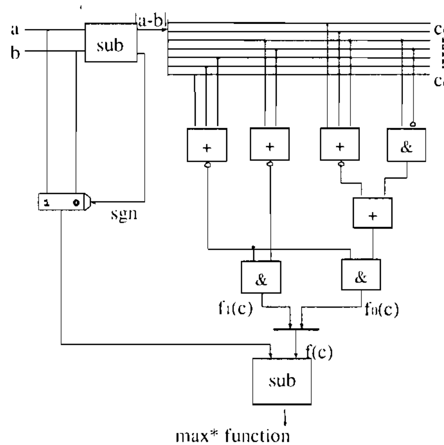
For the implementation of the calculations in the MAP-algorithm, two issues should be investigated in more detail. First, a correction function is needed for calculations in the log-domain, as explained in Section 2.5.2. Second, the α- and β values, calculated by recursion, can become very large: a normalization scheme is needed here.

Logarithmic correction function

As explained before, the log-MAP-decoder has a maximum function (max*) that consists of a maximum operation and a correction function (equation 2.14). The implementation of the correction function is not complex because fixed-point values are used. Figure 3.2 shows the correction function for floating- and for fixed-point values.



- Figure 3.2: the logarithmic correction function



- Figure 3.3: architecture for implementing the max* function

The implementation of the fixed-point max* function can be described as follows:

```
c = b - a;

if (c < 0)              c = (-c);
if (c > 2)              f(c) = 0;
else if  (c > 0.75)     f(c) = 0.25;
else if  (c > 0)        f(c) = 0.5;
else                    f(c) = 0.75;

if ( a < b)  max* = a – f(c);
else  max* = b – f(c);
```

Figure 3.3 illustrates the architecture used for implementing the max* function. We have assumed 7 bits for the implementation of *c*, from which two bits are used after the decimal point. *f(c)* consists of two bits: $f_1(c)$ and $f_0(c)$. These bits are described as:

$$f_1(c) = \bar{c}_6 \cdot \bar{c}_5 \cdot \bar{c}_4 \cdot (\bar{c}_3 \cdot \bar{c}_2) \qquad (3.1)$$

$$f_0(c) = \bar{c}_6 \cdot \bar{c}_5 \cdot \bar{c}_4 \cdot (\bar{c}_3 \cdot c_2 + \bar{c}_2 \cdot \bar{c}_1 \cdot \bar{c}_0) \qquad (3.2)$$

Normalization

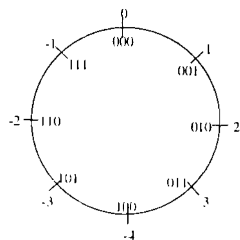
The second calculation issue that needs to be investigated is the normalization. The normalization of the metric calculation, needed to limit the number of bits in a metric word, relies on two properties: the output of the MAP-algorithm only depends on differences between metrics; and the difference between metrics is bounded. The two most common techniques for normalizing the metric calculation are:

- subtracting the largest state of a metric from the other states of the metric
- using two's complement arithmetic

Before investigating these methods, it should be noticed that, due to the recursive dependency within the decoding algorithm, calculations in one trellis step have to be conducted before the next step can be done. Additional calculations for the normalization consequently increase the latency within the decoder.

The first technique, *subtracting the largest metric from the others*, is rather straightforward but adds some operations to the recursive path: comparisons to estimate the largest state of a metric plus a subtraction for every state.

The normalization technique using *two's complement arithmetic* [Lit. 10] does not avoid overflow, but accommodates overflow in such a way that it does not affect the correctness of the results. In a common realization of two's complement arithmetic with *n* bits, both addition and subtraction are defined modulo 2^n. Due to this modulo operation, all values will lie on a circle. Figure 3.4 shows this circle for a 3-bit word.



• Figure 3.4: example of metric representation

If the maximum difference between two values in two's complement arithmetic is less than half a circle, it is always possible to determine which value is the largest one. For example:

(+3)-(+1)=011+111=010=positive → (+3) is the largest value
(-1)-(-4)=111+100=011=positive → (-1) is the largest value
(-4)-(+2)=100+110=010=positive → (-4) is the largest value

Note that because the maximum difference is less than four (half a circle), -4 is larger than 2.

This technique needs no calculations for normalization, so the decoding latency is not increased by the normalization. However, additional calculations are needed for the sign extension required in the soft output calculation (direct addition of metrics will exceed the maximum difference of half the cycle, loosing the property of knowing which one is the largest). Another disadvantage of this technique is, as simulations have shown, that three extra bits are needed for each metric word. One bit is needed for the sign extension. The need of the two other bits can be explained by the fact that they are necessary to cover the whole range of values that the metrics can have in the loss-less representation. This is not needed when subtractive normalization is applied because the metric values are saturated then: the maximum state is then subtracted from the outcome of the metric calculation.

For the basic implementation, we assumed that the normalization was done by using subtractive normalization. The implementation of the subtractive normalization is given in Appendix C.

Appendix C gives also the basic implementation of the metric-, extrinsic- and channel calculations, which we can derive from the MAP-algorithm.

Number of operations, memories and memory accesses

From the MAP-algorithm we can derive the number of metric-, extrinsic- and channel calculations that we need. This gives the following results:

- Metric calculations: $2 \cdot frame_size \cdot 2 \cdot number_of_iterations$
- Subtractive normalization: $2 \cdot frame_size \cdot 2 \cdot number_of_iterations$
- Extrinsic calculations: $frame_size \cdot 2 \cdot number_of_iterations$
- Channel calculations: $2 \cdot number_of_states \cdot frame_size \cdot 2 \cdot number_of_iterations$

One metric calculation includes the calculation of all states in that metric. The implementation of the calculations is given in Appendix C. Every single calculation consists of the following operations:

- Metric calculation: 12 additions, 8 max* operations
- Subtractive normalization: 7 comparisons, 8 subtractions
- Extrinsic calculation: 20 additions, 14 max* operations, 1 subtraction
- Channel calculation: 1 or 2 additions

For estimating the number of memories, each array is supposed to be a memory. Within the MAP-algorithm, we have 5 arrays: 2 metric arrays in which we store the recursive values and three input arrays: the systematic input array, the parity input array and the a-priori array. We have deducted the number of maccesses to these arrays from the MAP-algorithm itself:

- Metric array (for metric- and extrinsic calculation and subtractive normalization):
  $(2 \cdot 6 \cdot number_of_states + 16) \cdot frame_size \cdot 2 \cdot number_of_iterations$
- Syst-input array (for metric calculation):
  $(2 \cdot number_of_states) \cdot frame_size \cdot 2 \cdot number_of_iterations$
- Parity-input array (for metric- and extrinsic calculation):
  $(2 \cdot number_of_states + 4) \cdot frame_size \cdot 2 \cdot number_of_iterations$
- A-priori array (for metric- and extrinsic calculation):
  $(2 \cdot number_of_states + 1) \cdot frame_size \cdot 2 \cdot number_of_iterations$

The number of states is eight (see also Section 2.5.1). The frame-size we have chosen is 400 and the assumed number of iterations is six. As can be concluded from the derivations above, the number of iterations will have no impact on the relative energy- and area consumption for the basic implementation.

Estimating costs in terms of area and energy for memories and calculations

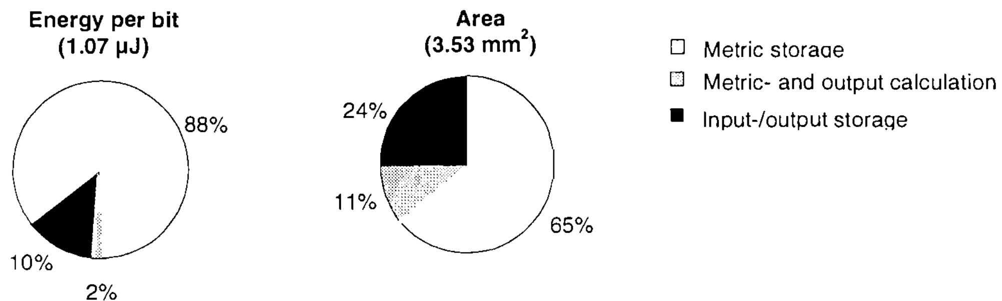
We have estimated the energy- and area consumption using the 0.35-µm CMOS process of Alcatel Microelectronics. For the estimations, the word-lengths of alphas, betas, channel- and extrinsic information have to be known. After simulations with a fixed point C-code, we observed no significant performance degradation for 7 bits for alphas and betas, 6 bits for the extrinsic information and 4 bits for the channel information.

The costs of the memories in terms of area and energy are given respectively by the formulas A.2 and A.1 in Appendix A. These formulas do not include the wire loads of the memories. We approximated the energy consumption due to these loads as can be seen in Appendix A.

We have approximated the costs of the arithmetic operations in terms of energy and area by using a tool called SYNOPSIS (this tool simulates the hardware layout for a given VHDL code). The costs in terms of area are directly given by

SYNOPSIS. However, the costs in terms of energy cannot directly be given because the switching activity of a single computation block is not known. To get an estimation of the energy consumption, an activity of 30% is typically assumed. The energy- and area consumption of the computation blocks is given in Appendix A.

Having investigated the three items that are necessary in order to estimate the energy- and area consumption, we can approximate the energy- and area consumption within the MAP-decoder. Figure 3.5 lists some indicative figures.



• Figure 3.5: relative area and energy consumption in the convolutional turbo decoder

3.2.2 Estimations for throughput and latency

In order to identify the bottlenecks in the implementation of the MAP-algorithm, we not only need estimations for the energy- and area consumption, but also estimations for the throughput and latency. The estimations for the throughput and latency will be investigated in this section.

The *throughput* and *latency* of the decoder depend on three factors: the delay caused by the interleaver operation, the number of iterations of the MAP-algorithm within the decoder and the inherent decoding latency in MAP-algorithm itself. The latency of the interleaving operation is assumed to be zero (see also Chapter 7). The number of iterations is assumed six. This iteration-number can be limited by using an early stop criterion [Lit. 17]. However, we did not regard a stop criterion in this thesis. The delay caused by the MAP-algorithm itself is investigated below.

The metric calculation forms the critical path within the MAP-algorithm. For the Alcatel 0.35-µm technology, the length of the clock-cycle turned out to be 13 ns after simulations with SYNOPSIS, which implies a clock-frequency of about 77 MHz (see Appendix A). Given the clock-frequency, the throughput and latency can be estimated when the number of clock-cycles that are needed to decode one frame is known. All accesses to the metric memories that are needed for the calculation of one single alpha or beta require one clock-cycle. Since the alpha- and beta calculations are recursive, all metric values within one frame have to be calculated serially. The total number of clock-cycles needed to decode one frame is 38400 (see also Appendix C):

- 48 clock-cycles are needed to calculate one metric consisting of eight states (four memory-reads and two memory-writes to the metric-memory for eight states of the metric).
- One half iteration contains 800 metric calculations (400 for the forward recursion and 400 for the backward recursion).

Given the clock-frequency and the number of clock-cycles within one frame, we have estimated a maximum throughput of 804 kbit/s and a minimum latency of 5.9 ms when applying twelve half iterations in pipeline. When no pipelining is applied, the throughput decreases to 67 kbit/s, while the latency stays the same.

3.2.3 Overview of bottlenecks in the MAP-decoder

From the estimations for throughput and latency and from Figure 3.5 it can be concluded that the area consumption is not a bottleneck for the implementation of the MAP-algorithm. However, the throughput and latency are respectively too low and too high for applications like WLAN (36 Mbit/s and 16 µs are needed respectively). Moreover, the energy consumption per bit is a bottleneck if higher bit-rates are achieved.

The estimations for throughput, latency and energy consumption show that the bottlenecks are mainly caused by the data-flow dominated recursions in the MAP-algorithm. This implies that much can be gained if the data- transfer and storage is optimized. Therefore, we applied a data transfer and storage exploration methodology developed at IMEC. The optimizations that we have applied by using this methodology are described in the next chapter.

4 Optimizations for the convolutional turbo decoder

The analysis of the basic implementation of the MAP-decoder has indicated a bottleneck in memory accesses. Thus, the memory architecture deserves a careful analysis in the design process. Although considerable effort has already been spent on the implementation of the turbo decoder ([Lit. 10] and [Lit. 11]), memory organization has not been explored in depth up to now. For that reason, we have applied a systematic Data Transfer and Storage Exploration (DTSE) methodology, developed at the Inter-university Micro-Electronic Center (IMEC) [Lit. 5], to the MAP-algorithm. The goal of this methodology is to optimize the execution order of the data transfers, in combination with the memory architecture. This reduces the energy consumption and the latency of the decoder. Furthermore, a higher throughput can be achieved.

Whether an optimization possibility exploited by the DTSE-methodology is beneficial for implementation (in terms of energy, area, throughput or latency) most often depends on the process technology that is used. In this report, the 0.35-µm CMOS process of Alcatel Microelectronics is assumed. All approximations for energy, area and timing have been estimated in the same way as for the basic implementation in Chapter 3.

If an optimization possibility turns out to be constructive, a trade-off still has to be made between energy, area and timing. Since the latency and throughput of the decoder are the main obstacles to applications like WLAN, we considered timing as the most significant factor. A high-throughput also means that the energy per decoded bit has to be minimized in order to get reasonable power consumption.

This chapter describes the application of the DTSE-methodology to the key-building block of convolutional turbo codes: the MAP-decoder; but first a brief introduction to the DTSE-technique is given.

4.1 Data Transfer and Storage Exploration methodology

The DTSE-methodology allows to systematically reduce the storage bottleneck in data dominated algorithms such as the MAP-algorithm. The methodology consists of several steps:

1. Global loop transformations are carried out to reduce size of the system-level buffers caused by long delays between the production and the consumption of a signal;

2. A memory hierarchy is defined in order to benefit from the available temporal locality in the data accesses: frequently accessed data can be read from smaller, less energy consuming, memories;

3. Memory units are allocated and arrays are assigned to these memories, taking into account the conflicts between arrays;

4. Data-flow transformations, such as modifying the computation order, shifting delay-lines through the algorithm or recalculation, are used to optimize energy- and area cost directly. Data-flow bottlenecks are removed as far as possible;

The DTSE-methodology normally applies data-flow transformations in the first step. However, in the data-flow transformations for the MAP-algorithm calculations will be traded off against memory accesses. In order to make

appropriate decisions for these transformations, the memory architecture has to be known. For this reason, we investigate data-flow transformations in the last step here.

4.2 Global loop transformations

As explained before, the throughput and latency are the main issues for the implementation of the turbo decoder. In order to optimize the throughput and latency, we have applied global loop transformations to the MAP-algorithm. The goal of these loop transformations is to parallelize the decoding process. This section describes the general method for parallelizing the MAP-algorithm.

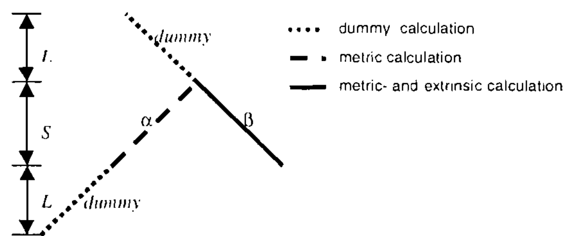
4.2.1 General strategy

In order to parallelize the decoding process, a subdivision of the MAP-algorithm into smaller independent pieces is needed. The main problem in parallelizing the MAP-algorithm is the recursion of the state metrics α and β: α_{N-1} recursively depends on all α_t, $t \in [0,N-2]$; β_1 depends on all β_t, $t \in [2,N]$. A modification of the MAP-algorithm is necessary to remove these dependencies.

4.2.2 Dummy metrics

The main idea of breaking the recursion sequence into several independent pieces is the assumption that the metrics are a function of only the previous L recursion steps. Any metric more than L steps before the current metric in the recursion has negligible influence on the decoding performance. Based on this idea, [Lit. 13] and [Lit. 16] introduced sub-blocks of the MAP-algorithm, which are called *windows*. The MAP-algorithm containing windows has been named *Overlapping Sliding Window* (OSW) algorithm since these windows overlap each other.

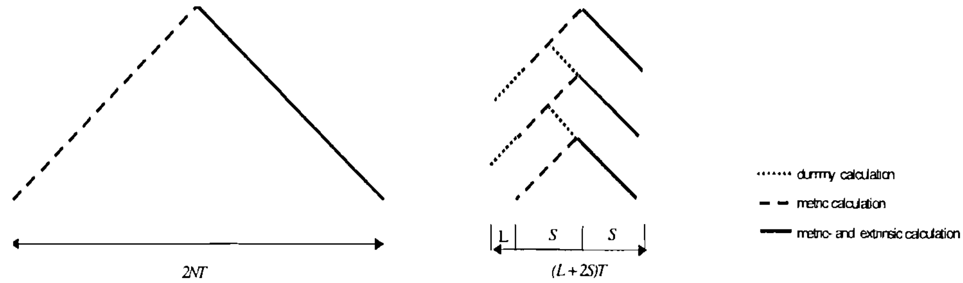
A window enabling parallelization is pointed out in Figure 4.1. The horizontal direction indicates the time, whereas the vertical direction indicates the recursion- or output calculation of a certain bit. The goal of the scheme is to calculate output without starting the recursions at the beginning or at the end of the trellis: there are no known α- or β state metrics in this sub-block. The so-called *dummies* are initialized with uniform values because all states are equally likely to have been visited during the coding process. After the initialization of the dummies, L recursion steps are carried out in order to obtain approximately correct values for the initial α and β. Given these values, reliable values for α and β can be calculated, each during S steps.



• Figure 4.1: window for enabling parallelization

4.2.3 Overlapping windows

Using the approach with dummy metrics, an arbitrary number of windows can be processed in parallel. Figure 4.2 compares the parallelized MAP-algorithm with the standard version.

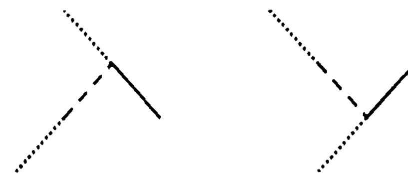


• Figure 4.2: comparison between standard an parallelized MAP-algorithm

The decoding latency $D = 2NT$ of the standard algorithm depends on the frame-length N. The latency in the parallelized version, however, depends on the length of the sequence of valid metrics calculated ($2S$) and length of the sequence of dummy metrics (L). The latency of the parallelized version consequently is $(L + 2S)T$. S can be varied to optimize the timing constraints. Simulations have shown that for a dummy sequence length $L \geq 18$ the degradation in performance is negligible.

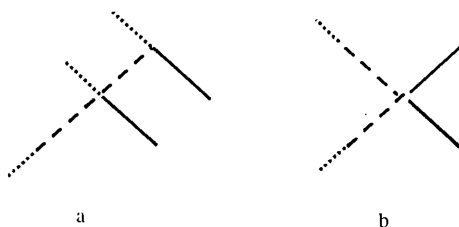
4.2.4 Workers

The parallelization introduced in the previous section uses windows that first calculate the α metrics, then the β metrics. Since the computation of α and β metrics is analogous, it is also possible to construct a window that first calculates the β metrics, then the α metrics. Both possibilities can be found in Figure 4.3. These windows are the smallest sub-blocks that can exist in the parallel decoding of a complete block.



• Figure 4.3: : two basic windows

The two basic windows can be combined in more than one way. Several combinations of windows are introduced in [Lit. 14] as *workers*. Some examples of these workers are shown in Figure 4.4. Some workers have time-slots in which valid α- and β-values are calculated simultaneously. Therefore, these metric values are being stored in different memories.



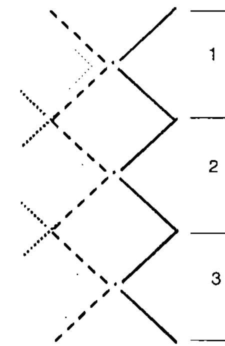
• Figure 4.4: different workers

Every worker relies on an interaction between several windows. Because of this, not every dummy sequence has to be calculated (see gray dots in Figure 4.4). Furthermore, some workers make memory reuse possible. This applies for worker *a* in Figure 4.4: when the first window is producing output, it consumes α metrics that are no longer needed.

The second window can then use this memory space. There is, however, also a disadvantage to the application of worker *a*: the decoding latency will increase because the second window starts later. Therefore, a trade-off has to be made here between energy consumption, throughput and latency.

4.2.5 Decisions for the implementation

As explained in the previous section, parallelization of the MAP-algorithm can be implemented by using workers. However, in the selection of a worker a trade-off has to be made between energy consumption, throughput and latency. Since throughput and latency are the most important issues, we have chosen the worker with the smallest latency and highest bit rate, as depicted in Figure 4.4b, for the implementation. Figure 4.5 shows three of these workers in parallel. As we can see, the worker size is twice the window size.



• Figure 4.5: chosen worker for parallelization

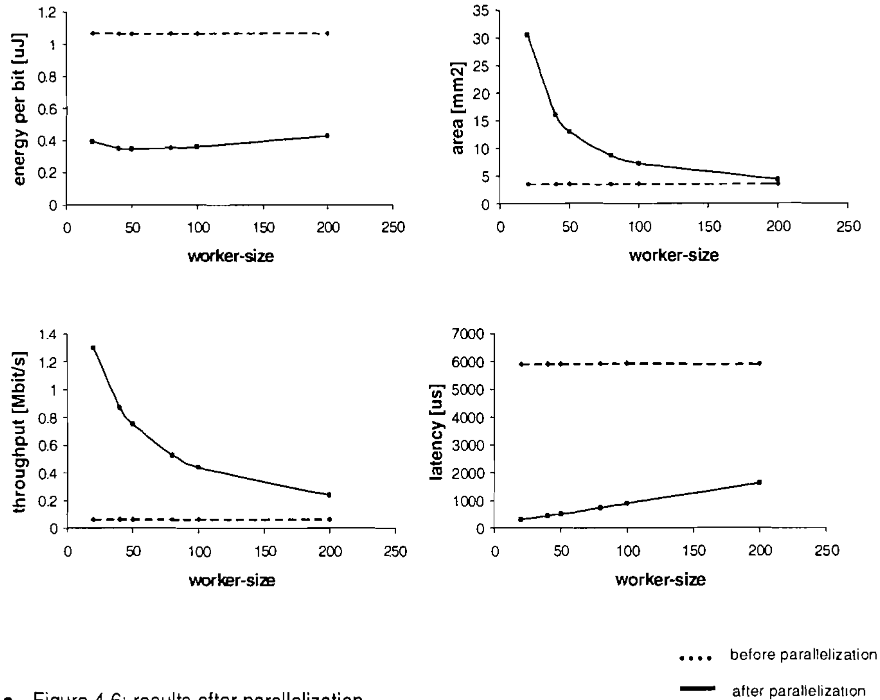
The maximum worker-size is 400, since the frame-size is assumed 400. The minimum worker-size depends on two items. First, the frame-size must be dividable by the worker-size. Second, applying a dummy-size of 18, the worker-size should not be smaller than 18. The reason for this is that we do not want to store input values double: the input values that are available in a worker for calculating valid metrics should also be used for calculating the dummies. This can not be done when the dummy-size is larger than the worker-size. Considering this, the minimum worker-size is set to 20.

Figure 4.6 shows the decoder parameter results after parallelization; the dashed lines indicate the results for a non-optimized MAP-decoder, the solid lines indicate the results after parallelization. These results have been estimated in the same way as the results for the basic implementation, see Appendix D.

Figure 4.6 shows that the energy consumption is reduced by a factor of about three. This reduction is due to the smaller memories used in the workers. A minimum for the energy consumption appears for a worker-size of about 50: for smaller worker-sizes, more dummies have to be calculated, whereas for larger worker-sizes, the size of the memories becomes larger and thus more energy consuming.

The area consumption is higher for all worker-sizes, as can be seen in Figure 4.6. However, the increase of the area consumption is much smaller for larger worker-sizes than for smaller ones. This is because of two reasons. First, many small memories consume more area than a few large ones. Second, the area for the arithmetic operations is multiplied by the number of workers.

The throughput is increased, as expected, for all worker-sizes. For smaller worker-sizes, more workers and more parallelization have been applied, which means an increase in throughput. For the same reason, the latency has decreased more for smaller worker-sizes.



• Figure 4.6: results after parallelization

After this first optimization step, the parallelization of the decoding algorithm, we will explore the second optimization step: introducing a memory hierarchy.

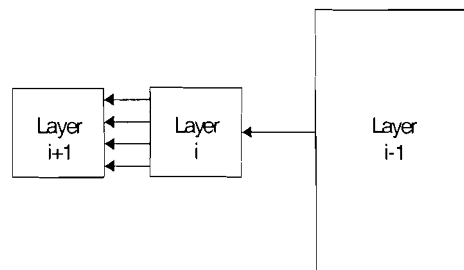
4.3 Memory hierarchy

We have defined a memory hierarchy for the MAP-decoder in order to benefit from the temporal locality in data accesses. This section first describes the principle of a memory hierarchy; after that, several possibilities for the memory hierarchy of the MAP-decoder will be exploited.

4.3.1 Principle of memory hierarchy

The energy consumption within a memory heavily depends on the size of the memory. In addition, smaller memories are closer to the data paths, thereby reducing the dissipation in the wiring. Introducing a memory hierarchy can reduce the memory size.

A first possibility for introducing a memory hierarchy is called *data-reuse*. The idea behind data-reuse is to store frequently accessed data into small memories so that afterwards, it can be read from these smaller memories. Applying data-reuse requires architectural transformations that consist of adding layers of smaller memories to which frequently used data can be copied. This principle is



• Figure 4.7: principle of data-reuse

depicted in Figure 4.7. Adding a layer involves a trade off: on the one hand, energy consumption is decreased because data is now mostly read from the smaller memory, while on the other hand energy consumption is increased because extra memory transfers are introduced.

A second possibility to introduce a memory hierarchy occurs when data that has been calculated is needed right after the calculation: it usually makes sense to store this data into smaller memories (caches) that are closer to the data path. However, when this data is also needed later in the algorithm, it has to be written to the main memory as well. This involves a trade off: on the one hand the read access right after the calculation will consume less energy when using a smaller memory, but on the other hand an extra write-access to this memory is introduced.

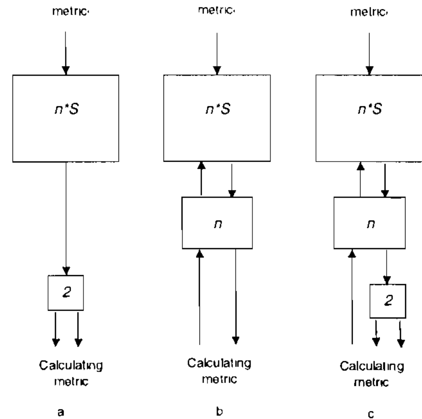
4.3.2 Memory hierarchy for the MAP-decoder

In this section, several possibilities for a memory hierarchy of the MAP-decoder will be explored. Therefore, we will investigate the data transfers to the five arrays of the MAP-decoder: the two metric arrays (α, β), the systematic input array (y^s_t), the parity input array (y^p_t) and the a-priori array (Λ^s_t). The size of the main memories of the α- and β-array is equal to the number of states multiplied by the window-size ($n*S$); the size of the main memories of the input arrays is equal to the window-size (S) itself.

Memory hierarchy possibilities for metric arrays

In order to explore optimization possibilities for the metric arrays, we have first investigated the accesses to the metric arrays that are needed for the metric calculations plus subtractive normalization. The metric calculations plus normalization can be written as a function of the metric arrays in a pseudo C-code (see also Appendix B.3):

```
for(t=0;t<S;t++)
{
  metric_t+1[0] = F(metric_t[0], metric_t[4]);
  metric_t+1[1] = F(metric_t[0], metric_t[4]);
  metric_t+1[2] = F(metric_t[1], metric_t[5]);
  metric_t+1[3] = F(metric_t[1], metric_t[5]);
  metric_t+1[4] = F(metric_t[2], metric_t[6]);
  metric_t+1[5] = F(metric_t[2], metric_t[6]);
  metric_t+1[6] = F(metric_t[3], metric_t[7]);
  metric_t+1[7] = F(metric_t[3], metric_t[7]);

  metric_min = metric_t+1[0];

  for(l=0;l<n;l++)
    if metric_min > metric_t+1[n]
        metric_min = metric_t+1[n];
  for(l=0;l<n;l++)
    metric_t+1[n]= metric_t+1[n]- metric_min;
}
```



- Figure 4.8: possible memory hierarchies for metric

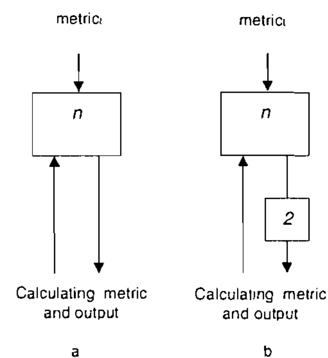
This code shows that each state of $metric_t$ ($metric_t[s]$) is used twice for calculating the next metric (see also Appendix B.3). Copying each state to a smaller memory is a possibility for data-reuse. Figure 4.8a represents this principle.

Another optimization possibility lies in the recursive calculation of the metric: the values of all states of $metric_t$ are needed to calculate the values of $metric_{t+1}$. Consequently, it is possible to store $metric_t$ in a cache and read from this cache to calculate $metric_{t+1}$. Figure 4.8b illustrates this possibility. Besides the saved memory accesses to the main memory for the calculation of the metrics, the normalization can also be calculated by accessing this cache only.

If both optimization possibilities turn out to be useful, the data-reuse of one metric state can be transferred to a lower level: the different values of $metric_t$ do not have to be read from the larger metric array then. Figure 4.8c shows this option.

The optimization possibilities in Figure 4.8b and Figure 4.8c can be extended if the output is calculated immediately after the calculation of the *metric.* The pseudo C code below illustrates this computation.

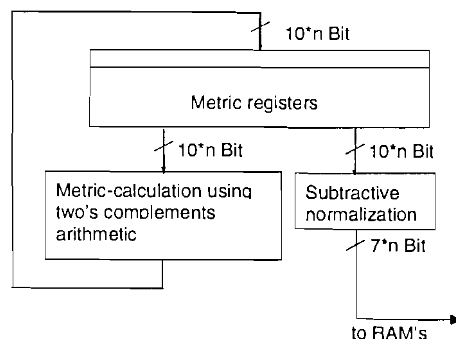
```
for(t=S;t<2S;t++)
{
  metric_t+1[0] = F(metric_t[0], metric_t[4]);
  metric_t+1[1] = F(metric_t[0], metric_t[4]);
  metric_t+1[2] = F(metric_t[1], metric_t[5]);
  metric_t+1[3] = F(metric_t[1], metric_t[5]);
  metric_t+1[4] = F(metric_t[2], metric_t[6]);
  metric_t+1[5] = F(metric_t[2], metric_t[6]);
  metric_t+1[6] = F(metric_t[3], metric_t[7]);
  metric_t+1[7] = F(metric_t[3], metric_t[7]);

  metric_min = metric_t+1[0];

  for(l=0;l<n;l++)
   if metric_min > metric_t+1[n]
       metric_min = metric_t+1[n];
  for(l=0;l<n;l++)
   metric_t+1[n] = metric_t+1[n]- metric_min;

 output_t = F1(metric_t+1) - F2(metric_t+1);
}
```



• Figure 4.9: memory hierarchy possibilities for *metric* without storage

Since *metric* is directly used for calculating the output, it needs not to be stored in the main memory and thus accessing the highest level is not necessary any more. Figure 4.9 illustrates this. The same principle applies for the dummy calculations.

When using one of the memory hierarchies that stores all n states of a metric in a cache, an improved implementation of the normalization scheme is possible. The implementation that we introduce needs no extra clock-cycles for the subtractive normalization. The scheme supposes the second layers in the memory hierarchy to be registers. The implementation is depicted in Figure 4.10. It is a combination of the two normalization-schemes that have been introduced in Section 3.2.1: subtractive normalization and the normalization using two's complements arithmetic. The metric computation with normalization has been split into two clock-cycles: the two's complement arithmetic calculation is performed in one clock-cycle; the outcome of this calculation is normalized by subtraction in the second clock-cycle. In this second clock-cycle, the two's complement arithmetic calculation for the next metric can already be executed.



• Figure 4.10: architecture for normalization

This solution needs no extra time for the normalization because of the two's complement arithmetic calculations. Moreover, it only needs three bits more for the computation (not for storing the metric values in the memories) because the results of the subtractive normalization are stored. Since the results of the subtractive normalization are also used for the extrinsic calculation, no additional calculations for the extrinsic are needed.

The implementation of the metric calculation and normalization using the introduced memory hierarchy is given in Appendix C.5.

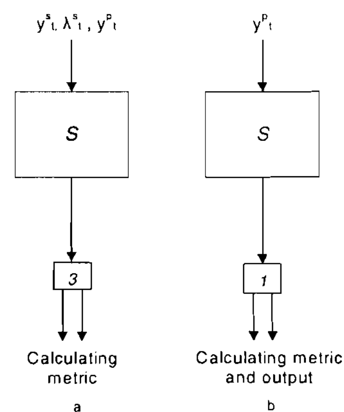
Having introduced a memory hierarchy for the metric arrays, we will now take a look at the possibilities for introducing a memory hierarchy for the input arrays.

Memory hierarchy possibilities for input arrays

As we know, there are three input arrays in the MAP-algorithm: the systematic input array (y^s_t), the parity input array (y^p_t) and the a-priori array (Λ^s_t). For investigating the optimization possibilities for these input arrays, we have examined the accesses to the input arrays that are needed to calculate the metrics and the output. These calculations can be written as function of the input-values in a pseudo C-code (see also Appendix B.4 and Appendix B.3). The piece of code given here, first calculates the α-metric and then the β-metric and output. Calculating the β-metric first will give the same possibilities for optimizations.

```
for(t=0;t<S;t++)
{
  αt[s] = F(y^s t-1+Λ^s t-1, y^s t-1+ Λ^s t-1 + y^P t-1, y^P t-1);
}
for(t=S;t<2S;t++)
{
  βt[s]= F(y^s t+1+Λ^s t+1, y^s t+1+ Λ^s t+1 + y^P t+1, y^P t+1);
  output t = F(y^P t);
}
```

In the code, $\alpha_t[s]$ stands for state s of α_t, which means that there are as much reads from the input arrays as there are states in order to calculate α_t. So, the first possibility for data-reuse shows up here: the element from an input value needed to calculate α_t can be stored into a cache of one word and accessed from here. Figure 4.11a illustrates this idea. A more careful look at the code shows that the same additions are used for every state of $\alpha_t[s]$. This means that these additions only have to be executed once to calculate one α-metric. The results of these computations can be stored in caches.



- Figure 4.11: possible memory hierarchy for inputs

A closer look at the code shows the possibility to use y^p_t for calculating both β and *output*. This idea is depicted in Figure 4.11b. In order to do this, the code has to be slightly changed: instead of calculating β_t, β_{t-1} should be calculated in the same for-loop as $output_t$ is calculated. Since β_{t+1} is needed to calculate $output_t$, there should be no problem to calculate the output. However, the β array will only be n large when the output is calculated right after it (see memory

hierarchy possibilities for metric array). Solving this problem can be done by exchanging the order of calculation for β and *output*:

```
for(t=S;t<2S;t++)
{
  output_t = F(y^P_t);
  β_t-1[s] = F(y^P_t);
}
```

4.3.3 Decisions for the implementation

As explained in the previous sections, five arrays can be exploited in order to introduce a memory hierarchy. In this section, we investigate whether the several possibilities that have been proposed in Section 4.3.2 are useful or not.

The decisions have been based on the 0.35-µm CMOS process of Alcatel. In this process, the minimum number of words in a memory is 16. Since the size of the caches is smaller than 16, we will use registers for implementing the caches. Multiplexers and de-multiplexers do not have to be included for addressing, since the values are always written to and read from the same locations.

For making decisions, the minimum worker-size of 20 has been chosen. If introducing a memory hierarchy is useful for this worker-size, it will definitely be useful for larger worker–sizes with larger memories.

Decisions for the memory hierarchy of the metric arrays

Table 4.1 shows the energy consumption of the memory accesses to the metric arrays for one metric calculation. The given optimization concerns the memory hierarchy depicted in Figure 4.8b.

	Energy consumption (µJ)
Before introducing memory hierarchy	0.0152
After introducing memory hierarchy	0.002476

- Table 4.1: energy consumption of the memory accesses to the metric-arrays for one metric calculation

As we can see in the table, the energy consumption can be reduced by a factor of six. Obviously, even more energy can be saved when no storage to the background is needed, as depicted in Figure 4.9. Since we used registers for this optimization, data-reuse of one state of a metric is not regarded anymore.

In addition to the energy reduction, the latency of the decoder will be reduced because of two reasons. First, the calculations can be done in parallel because all states of the previous metrics are available in the registers and can be read (and written) in one clock-cycle. Second, by applying the normalization scheme shown in Figure 4.10, no extra clock-cycle for the normalization is needed.

Decisions for the memory hierarchy of the input arrays

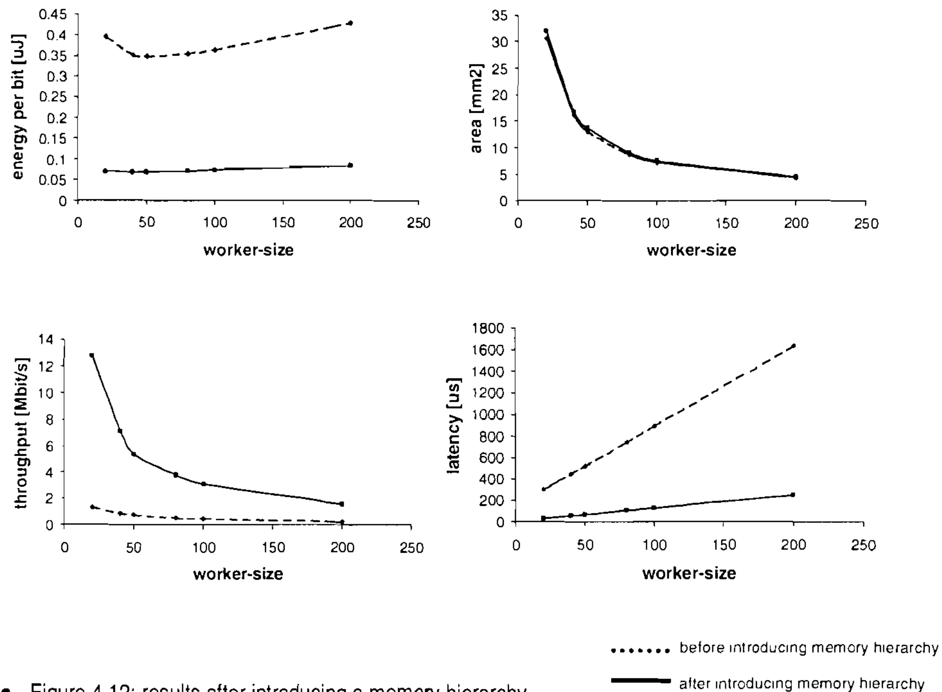
Table 4.2 shows the energy consumption of the memory accesses to the input-arrays y^s_t and Λ^s_t, for one metric calculation.

	Energy consumption (µJ)
Before introducing memory hierarchy	0.0089
After introducing memory hierarchy	0.0012

- Table 4.2: energy consumption of memory accesses to the input-arrays for one metric calculation

From this table, we can conclude that the reuse of the input values is beneficial for the energy consumption. Evidently, even more energy can be saved if the parity input, y^p_t, is not only reused for calculating metrics, but also for calculating the output: the data is already available in the cache.

Figure 4.12 shows the results after having introduced a memory hierarchy (see Appendix D for the estimations). The energy reduction is due to the applied optimizations; the small increase in area is because of the extra registers needed for introducing a memory hierarchy; the gain in throughput and latency is due to the parallel calculation of the metric and the optimized normalization scheme.



- Figure 4.12: results after introducing a memory hierarchy

Having applied the optimized the memory hierarchy of the metric arrays, the calculation of the metrics can be done in parallel. However, the storage of the states within one metric is still done serially. In order to store the states of a metric in parallel to the main memory, the allocation of the memories has to be optimized. This optimization will be discussed in the next section.

4.4 Memory allocation

The allocation of memories and the assignment of arrays to these memories give some further possibilities for optimizations. However, there is a restriction: arrays that are in conflict with each other (that means are being accessed at the same time) cannot be put into the same memory. These conflicts can be solved in several ways.

This section first investigates the conflicts between the several arrays for the main memories; after that, several solutions to solve these conflicts are given; finally, we will make decisions for the implementation.

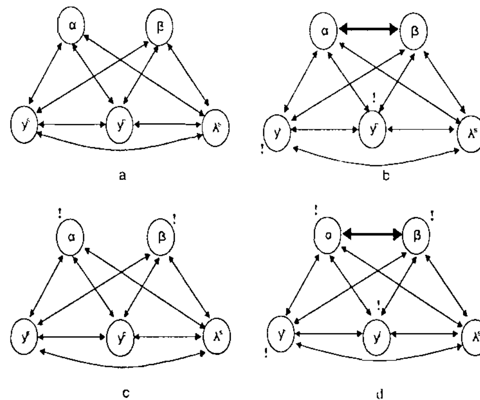
4.4.1 Conflicts between arrays in the MAP-algorithm

A conflict between two arrays exists if these arrays are being accessed at the same time. The number of conflicts between arrays mostly depends on the degree of parallelization (= calculation at the same time). In the previous sections, several kinds of parallelization have already been implemented in the MAP-algorithm:

- Parallelizing the MAP-algorithm by using workers (Section 4.2.4);
- Parallelizing the α- and β-calculations within one worker (Section 4.2.4);
- Parallelizing the calculations of the states within a metric (Section 4.3.3);

The first parallelization introduces conflicts between arrays in different workers, whereas the second one introduces conflicts between arrays within one single worker. Up to now, we have solved these conflicts by using different memories for these arrays. In the next section, some other possibilities for solving these conflicts will be explained. The last parallelization possibility does not introduce any conflicts between arrays to the main memory since it has been implemented by using caches (Section 4.3). However, at the end of the previous section, we noticed that not only the calculation of the metric calculations could be done in parallel, but also the storage of the states within a metric. This parallelization possibility will introduce conflicts in the main memories, though.

Conflicts between arrays can be illustrated in *conflict graphs*. The nodes in a conflict graph correspond to the arrays; an edge between two nodes indicates a conflict; an exclamation mark next to a node indicates a self-conflict in that array (a self-conflict means that different data items have to be read from the same array at the same time).



- Figure 4.13: conflict graphs for several degrees of parallelization

In the MAP-algorithm, we have five arrays: two metric arrays and three input arrays. Before allocating these arrays to memories, we first have to investigate the conflicts between these arrays. Figure 4.13 shows the conflict graph between these arrays within one single worker. The bold edges and the bold exclamation marks imply a difference from the condition where no parallelization is included (Figure 4.13a). Figure 4.13b represents the simultaneous calculation of the α's and β's, which causes self-conflicts within the input arrays and between the metric arrays. The self-conflicts to the input arrays are introduced because we need different values of these arrays at the same time now the α's and β's are calculated simultaneously. The conflicts between the metric arrays is evident: the α's have to be read (and written) at the same time as the β's. Besides calculating α's and β's simultaneously, we saw that it is also possible to store the states within one metric simultaneously (Figure 4.13c). This parallelization causes self-conflicts within the metric arrays because we have to read (and write) the several states

within a metric array at the same time. In Figure 4.13d, the combination of the parallelization possibilities is given (that is calculating α's and β's simultaneously and storing the states of one metric in one clock-cycle).

4.4.2 Solutions for array conflicts

As described in the previous section, arrays that are in conflict with each other cannot be put into the same memory. Several possibilities are available to deal with these conflicts:

- Using multi-port memories instead of single-port memories
- Using different memories for the arrays that are in conflict with each other
- Splitting the arrays with a self-conflict into parts that are not in conflict
- Storing the values that are needed at the same time in one word of a memory

Using *multi-port memories* is not recommended for two reasons. First, multi-port memories are not always available. Second, if these ports are available, they are very costly in terms of area and energy.

Using *different memories* for arrays that are in conflict with each other is the most straightforward solution. We have used this solution for solving conflicts up to now. However, using different memories should be avoided as much as possible for arrays with a self-conflict: using additional memories implies multiplying the number of memories with the number of self-conflicts.

Splitting the arrays with self-conflicts is obviously a better solution: the number of memories is still multiplied by the number of self-conflicts, but the size of the memories will be smaller. In the MAP-decoder, all self-conflicts can be solved by splitting the arrays. The input arrays can be split into a bottom- and a top part: the bottom part being used for the calculation of one metric, the top part for the calculation of the other metric. The metric arrays can be split into different arrays for the different states.

However, an even better solution for solving the self-conflicts of the metric arrays can possibly be found by *storing the values that are needed at the same time in one word of a memory*. This implies that by accessing one word of the memory, more values are accessed at the same time, which reduces the number of memory accesses. The same principle can be used for solving the conflicts between different input-arrays: values of these arrays that are needed at the same time can be stored in one word of a memory. Whether this solution decreases the energy consumption or not depends on process architecture that is used: although fewer accesses are needed, accessing a long word is more energy-consuming than accessing a short word.

4.4.3 Decisions for the implementation

In the decisions for the implementation, we have also to decide how the conflicts between arrays will be solved. The conflicts between the arrays when no parallelization is involved are shown in Figure 4.13a. We have solved the conflicts between the input-arrays by putting these arrays in one word of a memory; all other conflicts have been solved by using different memories for these arrays.

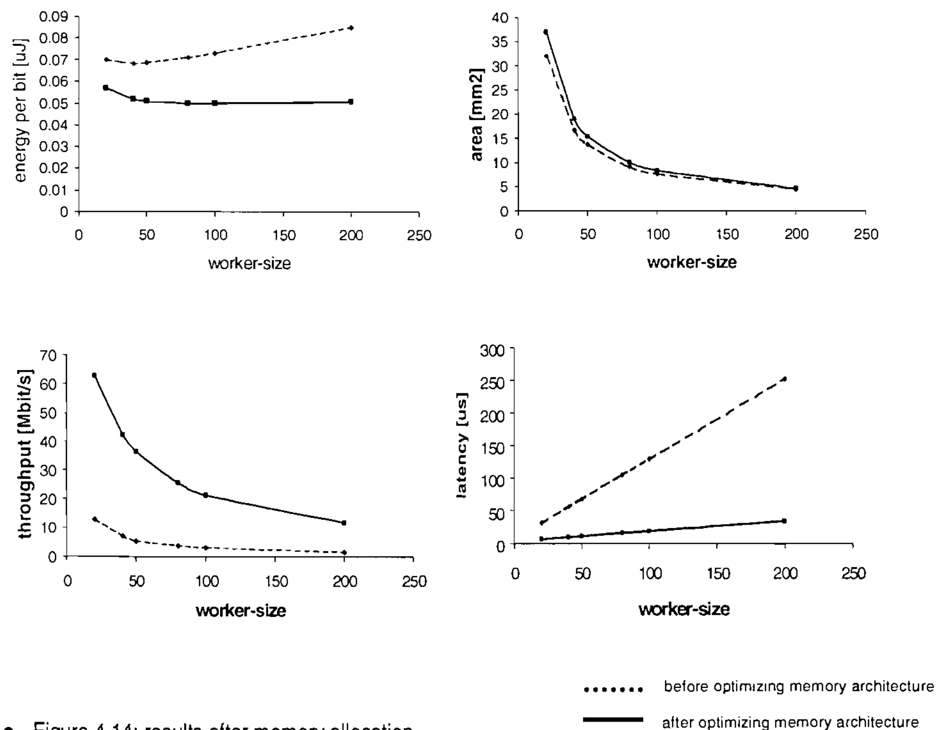
Section 4.4.1 listed three parallelization possibilities that introduce more conflicts.

The first parallelization possibility, to introduce workers, introduces conflicts between the arrays within the different workers. We have solved these conflicts by using different memories for the arrays within different workers.

The second parallelization possibility, to calculate α's and β's simultaneously, introduces self-conflicts to the input arrays and a conflict between the α- and β array. We have solved the first conflict by splitting the bottom and top part of the input-arrays and putting them into different memories. The second conflict we have solved by using different memories for the two metric-arrays.

The third opportunity for parallelization is to store the several states of one metric in parallel. This implies self-conflicts in the metric-memories because the different states within one metric have to be accessed at the same time. In order to solve this conflict, we have stored the values of the different states in one word of a memory. The maximum word-length of a memory in the Alcatel 0.35-µm CMOS process is 32. This means that a maximum of 4 different states (each having a word-length of 7 bits) can be stored in one word of a memory.

Figure 4.14 shows the results after memory allocation. The energy consumption has decreased due to the applied optimizations for the memory allocation. Using more memories and larger word-lengths for the metric memories causes an increase in area, but gains in throughput and latency. More details can be found in Appendix D.



- Figure 4.14: results after memory allocation

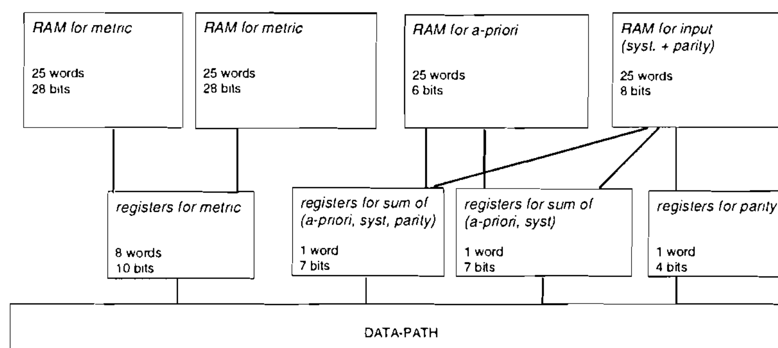
4.4.4 Memory architecture

Before applying the next optimization, data-flow transformations, we have to define the memory architecture, since calculations will be traded off against memory accesses. Therefore, we have to choose a worker-size. Figure 4.14 shows that for smaller worker sizes the throughput and latency improve, whereas the area consumption increases. For a worker-size of about 40, 50 the minimum

energy consumption can be found. Although throughput and latency are the most important issues, a worker-size of 50 has been chosen for three reasons. First, the achieved throughput is high enough for WLAN applications (36 Mbit/s). Second, the energy costs are minimal. Third, the area increases rapidly for smaller window-sizes.

Having defined the optimal worker-size, we know the size of the memories that is needed. Figure 4.15 represents the memories and registers that are needed within a one window of a worker. The applied worker consists, as said before, of two windows.

Two memories within one window are needed for the storage of the metric values, each storing 4 states of 25 metric values. Two other memories contain the input values: one memory the parity- and systematic information, the other memory the a-priori information. The registers for the metric values consist of 8 states of 10 bits: for calculating the metric values, 10 bits per word are needed. Two of the registers that store the input values need 7 bits because the values in these registers are additions of more input values; the other register only stores the parity information.



- Figure 4.15: memories within one window of a worker

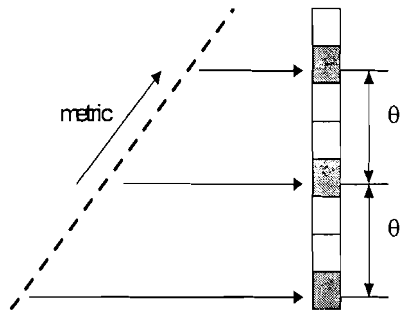
Knowing the memory hierarchy, we can make a trade off between memory accesses and calculation. This possibility will be investigated in the next section.

4.5 Data-flow transformations

As we have concluded in Chapter 3, the energy consumption of the metric memories is a large factor in the total energy consumption. This observation leads to the idea that not every intermediate result that will be used further in the algorithm should be stored. Whenever a value is needed that was not stored, it is recalculated from the remaining data and used immediately [Lit. 13].

4.5.1 Selective recalculation for the MAP-algorithm

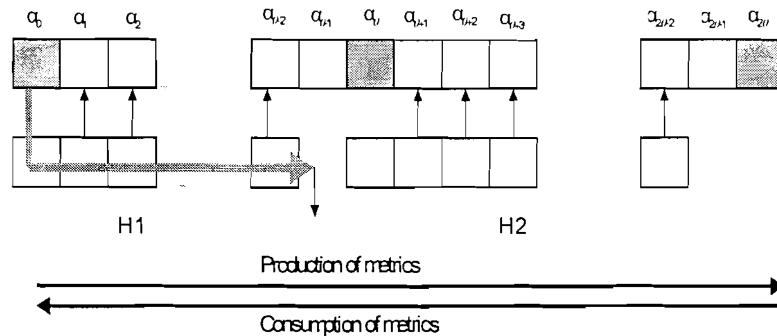
In the recursions of the MAP-algorithm, the size of memories and the number of accesses to these memories can be traded off against additional computations by only partially storing calculated state metrics. This is outlined in Figure 4.16.



- Figure 4.16: partial storage of metric

The idea is as follows: instead of storing every metric after each recursion step, the metrics are only stored every θ steps. If values that have not been stored are needed in the reverse recursion, they are recalculated. This recalculation uses a small cache to store the newly recomputed values. Since the cache is small, every access to it will consume less energy.

The top bar in Figure 4.17 represents a section of a sequence of valid α's. The direction of production of these α.'s (left to right) and their consumption (right to left) are indicated. The bottom bar in Figure 4.17 (*H1* and *H2*) represents two caches. Using these caches, the recalculation can now be scheduled as follows: during the production of the metrics, only α_0, α_θ, $\alpha_{2\theta}$ etc. are stored in memory. When the metrics are consumed in the backward direction, recalculation is necessary. Suppose *H2* is filled with the intermediate metrics $\alpha_\theta \ldots\ \alpha_{2\theta-2}$. When the production of extrinsic information consumes $\alpha_{2\theta-2}$, the recalculation of α_1 from α_0 is executed and α_1 is stored in *H1*. The next step consumes $\alpha_{2\theta-3}$ and recalculates α_2. After another θ-3 steps, α_θ is consumed en $\alpha_{\theta-1}$ is produced. This last metric can be buffered in a register, because it will be consumed in the next step, and the process of recalculation can restart on the next segment. Note that every time a cache is accessed to read a value, another value is written. *H1* and *H2* can thus be implemented on the same piece of memory.



- Figure 4.17: principle for recalculation

Given the parameter θ, the size of the memory needed for metric storage is divided by θ. In addition, a cache with $n(\theta-1)$ registers is necessary (n being the number of states). However, more calculations are needed when applying recalculation. So there is a trade off here: if the energy needed for the extra calculations and the cache accesses is less than the savings gained from the metrics storage, recalculation is useful.

4.5.2 Decisions for the implementation

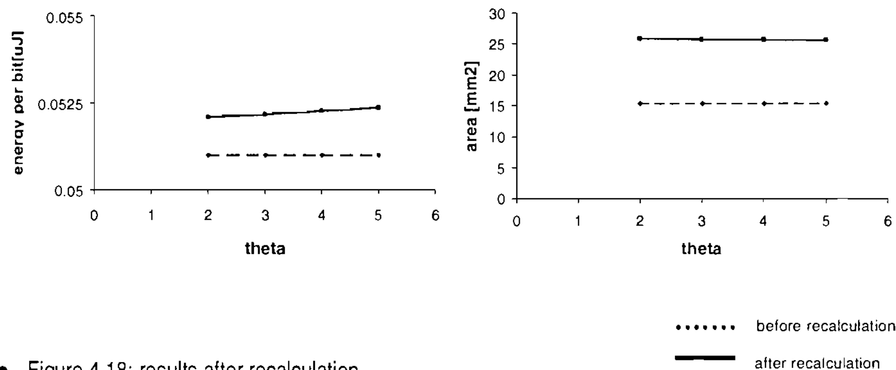
Having defined the size of the worker in Section 4.4.3, we can investigate the usefulness of selective recalculation. As explained in Section 4.5.1, the value of θ determines how many metric values are stored in the main memory and thus the size of this memory. For estimations for the energy- and area consumption, it should be noticed that the number of words within one memory cannot be smaller than 16 in the 0.35-µm model of Alcatel. This means that these memories should

be replaced by register-files. However, no reliable information is available for such files. Therefore, we have also based the estimations for smaller memories on the formulas A.1, A.2 and A.3.

After having applied recalculation, throughput and latency will stay the same. The energy- and area consumption is given in Figure 4.18. Although the number of accesses to the metric memories and the size of these memories have decreased, the total energy consumption has increased. This is mainly due to the extra calculations and the extra accesses to the input-memories and the cache memories needed for recalculation. Furthermore, as formula A.3 shows, the accesses to the smaller metric memories consume relatively much energy.

The increase in area is because of the double number of the input memories and the calculation units, which are needed in order to be able to calculate the metric values and the recalculated values in the same clock-cycle.

Since both energy- and area consumption have been increased, we will not implement recalculation.



- Figure 4.18: results after recalculation

4.6 Optimized convolutional decoder

In order to optimize the implementation of the turbo decoder, we have applied several optimizations. First of all, we have parallelized the MAP-algorithm. Then, we have introduced a memory hierarchy and optimized the memory architecture. These optimizations were beneficial for both timing and energy consumption. However, the last optimalization possibility, recalculation, increased the energy- and the area consumption, while the throughput and latency did not change. Therefore, we did not apply recalculation.

The results of the non-optimal and the optimal version for a worker-size of 50 can be found in Table 4.3.

	Energy (μJ/bit)	Area (mm^2)	Throughput (Mbit/s)	Latency (ms)	Power (W)
Non-optimized	1.07	3.53	0.07	5.9	0.075
Optimized	0.051	15.4	36.3	0.01039	1.85

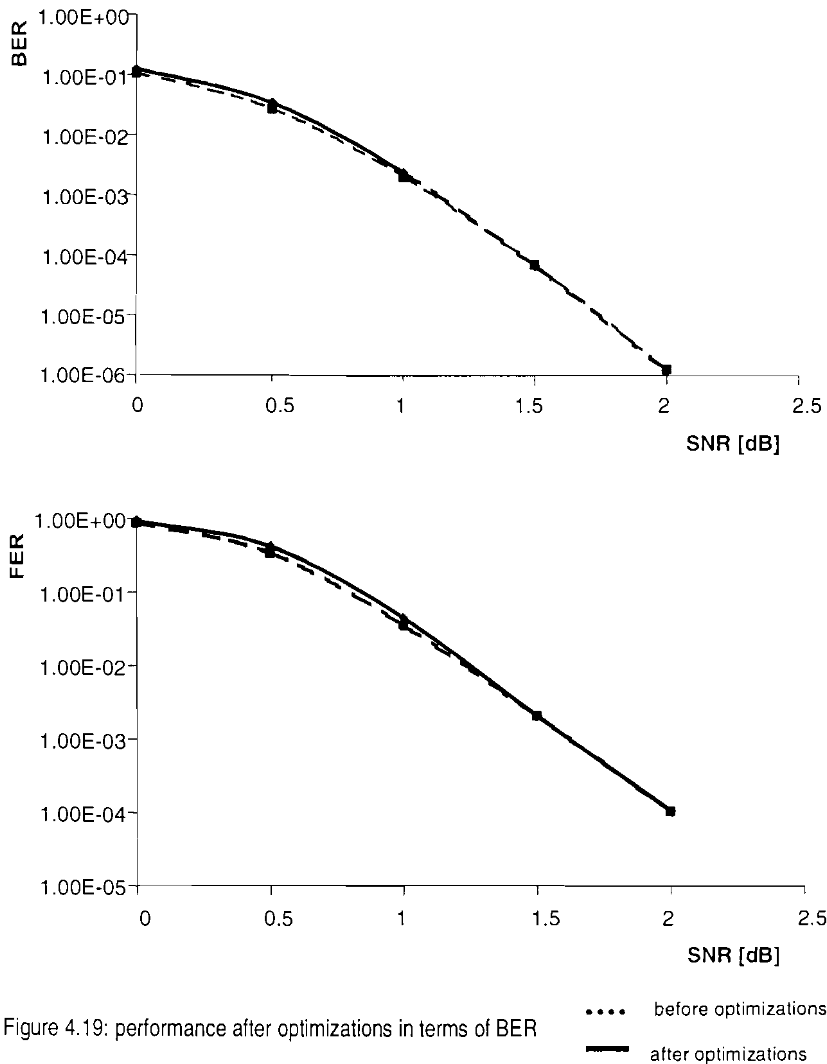
- Table 4.3: results comparing the non-optimized- and optimized- decoder

The applied optimizations show a 25-fold energy reduction per decoded bit, while at the same time, the speed is multiplied by 400 and the latency divided by 400. We have achieved these gains at the cost of the logic- and memory area: the total

area is increased by a factor of 5. The results show that it is possible to combine the good performance of turbo codes with an implementation that has a high throughput, a low latency and low power consumption. The optimized decoder has now the required throughput and latency (respectively 36 Mbit/s and 16 μs) for WLAN. In addition, the energy consumption per decoded bit is low enough to end up with reasonable power consumption.

The applied optimizations cause a negligible degradation in performance. This degradation is mainly caused by the introduction of the OSW scheme. A fixed-point C-code describing this architecture was written and simulations for an AWGN channel were performed.

Figure 4.19 shows the Bit Error Rate (BER) and the Frame Error Rate (FER) for the standard MAP and the final optimized version. The dashed lines represent the standard version, while the solid lines represent the optimized version. For the BER, a maximum performance degradation of 0.03 dB at a BER of 1.20E-02 can be noticed, while for the FER a degradation of 0.08 dB at a FER of 1.50-02 is observed.



- Figure 4.19: performance after optimizations in terms of BER

5 Product turbo code

In the previous chapters, we have investigated a convolutional turbo code. The encoder produces a convolutional of the input bits, whereas the decoder is based on iterations between single MAP-decoders that provide soft output information. This chapter describes the *product turbo code*. In this code, the encoder adds redundant bits to a data-block, resulting in a code word. The decoder of the product turbo code contains, as the convolutional turbo decoder, single decoder modules that produce soft output information for further iterations. The decoder modules themselves are, however, different to the MAP-decoder.

The performance of the product turbo code is quite similar to the performance of the convolutional turbo code [Lit. 22]. The implementation of the product turbo decoder will be briefly investigated in this chapter. No profound study has been done on the implementation because the available time was limited. Therefore, it is difficult to compare the implementation of the product turbo code with the implementation of the convolutional turbo code.

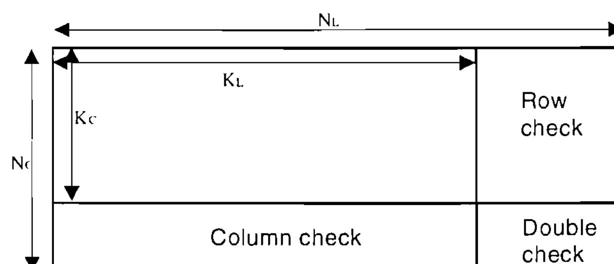
This chapter first describes the product turbo code technique; after that, the bottlenecks for the implementation of the product turbo decoder are defined; given these bottlenecks, some global optimizations for the implementation will be exploited; finally, the results of these optimizations are given.

5.1 The algorithm

The coding process of the product turbo code can be split into an encoding- and a decoding part. This section briefly describes both elements.

5.1.1 Encoding part

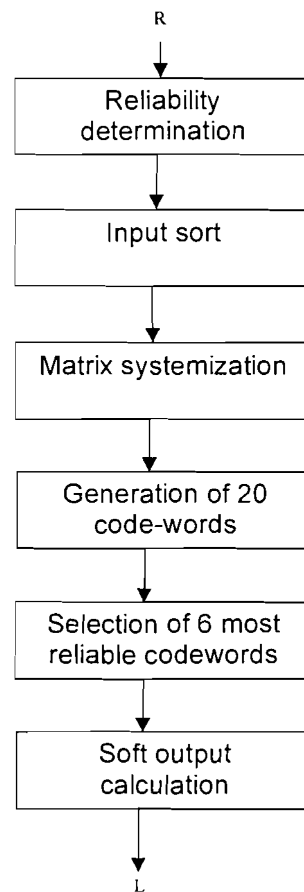
Figure 5.1 depicts the principle of the product encoder. The input of the encoder is split into blocks of length K_L. These blocks can be put in K_C rows, resulting in a matrix with K_L columns and K_C rows. At the end of each row and column, *parity check bits* are added to end up with a code word of length N_L, respectively N_C. In order to ad these parity check bits, a row or a column is multiplied with a so-called *generator matrix.* The first part of the generator matrix is an identity matrix. The generator matrix for a row consists of K_L independent vectors of N_L bits; the generator matrix for a column consists of K_C independent vectors of N_C bits. Linear combinations of these independent vectors can describe all code words.



• Figure 5.1: product encoding

5.1.2 Decoding part

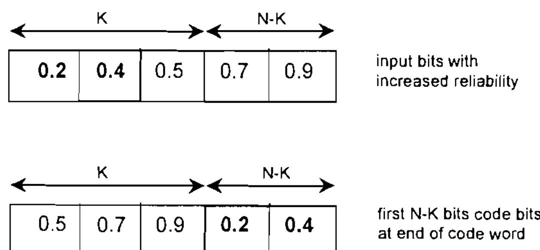
As we know, turbo decoding is an iterative process where two soft output decoders use each other's (inter)leaved information. In product turbo decoder we have investigated, the input values are not subtracted from the output as in the convolutional turbo decoder, but are part of the a-priori information for the next decoding step. A second remarkable difference with the convolutional turbo decoder is the interleaving process: we do not need separate interleaving blocks, but the interleaving is done by calculating the rows in one decoding step and the columns in the next decoding step. The number of iterations for the applied product turbo decoder is typically four.



- Figure 5.2: product decoding

Most product decoders are based on a trellis graph [Lit. 20]. However, the Fang Buda Algorithm (FBA) [Lit. 18] does not need a trellis graph for decoding the rows and columns, which reduces the complexity. In addition, the performance is quite similar to the performance of the convolutional decoder [Lit. 22]. Figure 5.2 presents the principle of this decoding algorithm. *R* represents the code word that has to be decoded, while *L* corresponds to the soft output of the FBA.

The first step in the algorithm determines the reliability of each bit in a code word by calculating the log-likelihood ratio (that is the probability that the received bit is a *zero* divided by the probability that the received bit is a *one*).



- Figure 5.3: soft output decoding

In the second step, the input bits are sorted with increased reliability. This sorting is done with the bubble-sort algorithm (Appendix G.1). After the sorting, the first *N-K* bits (that is N_L-K_L and N_C-K_C for respectively rows and columns) are put at the end of the code word. Consequently, these bits form the parity part of the new code word (see Figure 5.3).

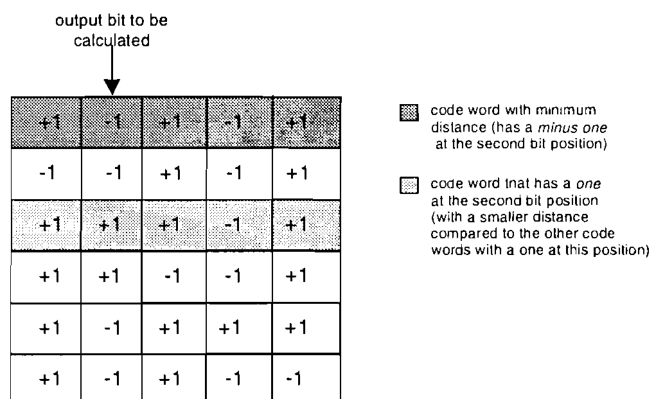
The matrix systemization rearranges the generator matrix corresponding to the sorted input. This is done in two steps: first, the columns of the matrix are put into the same order as the bits of the code word; second, this matrix is modified so that the first part will be an identity matrix again.

The systemized generator matrix is used for generating new code words out of the first *K* bits (that is K_L and K_C for respectively rows and columns) of the sorted input. In order get different code words, we need different combinations of these first *K* bits. In order to make these combinations, the hard-decision of some bits is changed: first of all the hard decisions with the least reliability. How the combinations of these first bits are exactly made has been described in [Lit. 18].

The number different combinations that we use for generating new code words is typically set to 20 (see also [Lit. 18]).

Having produced these 20 code words, the 6 code words with the minimum distance to the received code word are selected. In order to do this, the first six code words are stored in increasing-distance order. The bubble-sort algorithm is used for this sorting. All other code words are compared with their distance to the distance of the sixth code word. If the distance of such a code word is smaller than the distance of the sixth code word, this code word will replace the sixth code word. After that, the new list of six code words is sorted again.

Having six code words, the soft output for each bit is calculated. This calculation is done by selecting two code words with the minimum distance on the following conditions: one of these code words has a *one* at the specific bit position and the other code word has a *zero* at that bit position. Figure 5.4 shows this principle when the second bit has to be calculated. The subtraction of the distances gives the soft output for that specific bit (L). If all code words have a *one* or a *zero* at the specific bit position, a scaling factor is used for calculating the soft output L.



• Figure 5.4: soft output decoding

The soft output information that is sent to the next decoding iteration is equal to $R + a \cdot (R - L)$. In this formula, a depends on the number of iterations that has already been done: the more iterations, the more reliable the soft output L and the larger a will be. The values that can be chosen for a are given in [Lit. 18].

5.2 Optimizations for the product turbo decoder

Before investigating possible optimizations, the bottlenecks in the basic implementation should be identified.

5.2.1 Identifying the bottlenecks

Although a hardware implementation has not been examined yet, the costs in terms of energy and area for memory usage can be estimated using ATOMIUM [Lit. 19]. ATOMIUM is a tool that analyzes the memory usage of array data in C programs and identifies the memory-related bottlenecks by means of C code instrumentation and simulation. The output of ATOMIUM is given in Appendix E.

Given the number of accesses to each array, we can estimate the energy- and area consumption based on the 0.35-µm CMOS process of Alcatel. The energy consumption can be estimated using formula A.3; the area consumption by using formula A.1. The word-length of the soft decision values is assumed to be 5

[Lit. 18]. Table 5.1 gives the estimations of the energy- and area consumption in the memories for the non-optimized version.

	Energy per decoded bit (μJ)	Area (mm²)
Non-optimized	3.19	9.63

- Table 5.1: energy- and area consumption for the non-optimized product decoder

We can conclude from this table that the energy per decoded bit is too high for a high-throughput implementation. The area is not a bottleneck for implementation.

5.2.2 Optimizations

The FBA mainly consists of reordering data, which involves many memory accesses. Therefore, most of the applied optimizations for the implementation intend to reduce the number of accesses to the arrays or to reduce the size of the arrays. This section will describe the applied optimizations.

Using smaller memories

The most straightforward optimization for the implementation of the product turbo decoder is to use memories with the size of a single row or column instead of the size of the whole matrix. This can easily be done since every row and column can be decoded independently. Reducing the size of the memories is beneficial for the energy consumption. In addition, a gain in either area consumption or throughput can be made: on the one hand every code word can be decoded serially and thus area can be saved, while on the other hand the throughput can be increased by decoding every code word in parallel. An in-between solution is also possible. Since the throughput of the decoder has not been investigated yet, serial decoding is assumed.

Simplified reliability determination

For determining the reliability of each bit, it is not necessary to calculate the *LLR*. Assuming a BPSK-modulation and an AWGN-channel (both described in Chapter 2), we can the following formula for the *LLR* (see Appendix F):

$$\begin{aligned} LLR &= -\log \frac{p(R|L=-1)}{p(R|L=+1)} \\ &= \frac{1}{2\cdot\sigma^2}\cdot 4R \end{aligned} \tag{5.1}$$

In equation 5.1, the sign of the *LLR* determines the hard decision and its absolute value gives the reliability of the hard decision. A similar simplification can be done for other modulation schemes [Lit. 27].

Different sorting algorithm

The sorting algorithm, used for the input sort and the selection of six reliable code words, can be implemented more efficiently using Shell's method. In [Lit. 23] it is proved that this method is the most efficient sorting method for sorting small lists

(that are lists with maximum 50 elements). The C code for Shell's algorithm is given in Appendix G.2.

Optimized search for six code words

We can even further optimize the searching of the six code words. Instead of sorting the six code words, the index of the code word with the maximum distance can be stored. After having stored the first six code words and the index of the code word with the maximum distance, the other code words are compared with their distance to the code word with this index. If the distance of such a code word is smaller, this code word will replace the code word with the index. The distance of this code word has to be compared to the distance of the other five code words and the index of the code word with the maximum distance will be stored.

Thus, by using an index for the code word with the maximum distance, the sorting of the code words is not necessary any more.

Optimized soft output calculation

For the soft output calculation, it is necessary to check whether a bit in a certain position is opposite to the bit in the same position in the code word with the minimum distance. However, in the previous step all bits of the code words have already been checked. In order to avoid a second check, the bits in a code word can be labeled during the execution of the previous step: during the soft output calculation only this one-bit label has to be checked.

Although energy can be saved by using labels, the area will increase. However, the area consumption was not a bottleneck for the implementation.

5.3 Results

A comparison of the implementation before and after the first optimization is given in Table 5.2. The results of the optimized version have been estimated in the same way as the results of the non-optimized version. The output of ATOMIUM for the optimized implementation is given in Appendix E.

	Energy per decoded bit (µJ)	Area (mm^2)
Non-optimized	3.19	9.63
Optimized	0.6	3.73

- Table 5.2: results comparing the non-optimized and the optimized product decoder

As can be seen from the table, both energy- and area consumption have been decreased, respectively with a factor of 5 and a factor of 3. The decrease in energy consumption is due to all optimizations that have been applied. However, the energy per bit is still far too large for a high-throughput decoder. Chapter 7 will give some recommendations for further optimizations.

The decrease in area consumption is due to the use of smaller memories. Nevertheless, the gain in area consumption will decrease when the decoding process will be more parallelized.

The applied optimizations will not have any influence on the performance of the product turbo decoder, since the algorithm itself has not been changed.

As already mentioned, the performance of the product turbo code is quite similar to the convolutional turbo code. If we also want to compare the hardware implementation of the two decoders, we should be aware of the fact that we did not take into account the calculations for the product decoder. So, in order to make a fair comparison, either the calculations for the product turbo decoder have to be taken into account or the costs due to the calculations in the implementation of the convolutional decoder should be excluded. In [Lit. 22], a comparison is made where the costs of the calculations in the convolutional decoder are excluded.

6 Conclusions

In this project, we investigated the implementation of two types of turbo decoders for WLAN applications: the convolutional turbo decoder and the product turbo decoder.

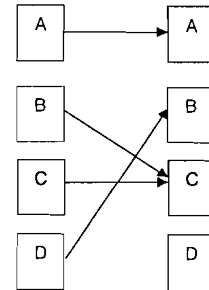
The basic implementation of the convolutional turbo decoder achieved a bit-rate of about 80 kbit/s. For WLAN, however, a throughput of about 36 Mbit/s is needed. Moreover, the energy consumption per decoded bit in the basic implementation was too high (1 µJ/bit) for a high-throughput decoder. In order to increase the throughput and decrease the energy consumption, we have applied several optimizations. These optimizations include parallelization, introducing a memory hierarchy and optimizing the memory architecture. In order to parallelize the decoding algorithm, the recursive dependency within the algorithm was broken by using dummies. After that, we introduced a memory hierarchy so that we could benefit from the locality of data. Finally, we optimized the memory architecture, taking into account the conflicts between arrays. Having applied these optimizations, we see for a certain degree of parallelization a 25-fold energy reduction per decoded bit, while at the same time the speed is multiplied by 400 and the latency is divided by 400. These gains have been achieved at the cost of logic- and memory area: the total area consumption is increased by a factor of five. The results of the convolutional turbo decoder show that its good performance can be combined with an implementation that has a high throughput (36 Mbit/s), a low latency (10 µs) and an acceptable power consumption (1.85 W). These estimations have been based on the 0.35-µm process of Alcatel. For an implementation on a 0.18-µm process, the results will even be better. The optimized version of the MAP-algorithm can thus certainly be used for WLAN applications.

For the implementation of the product turbo decoder, only the energy- and area consumption have been investigated. The energy consumption per decoded bit was far too large for a high-throughput decoder (3.19 µJ/bit). In order to reduce the energy consumption per decoded bit, we have applied some first optimizations. These optimizations include for example the usage of smaller memories and a different sorting algorithm. The applied optimizations have improved both energy- and area consumption with respectively a factor of five and a factor of three. However, the energy consumption per decoded bit is still far too large (0.6 µJ/bit) for the implementation of a high-speed turbo decoder. Therefore, further optimizations are necessary.

7 Future work

Interleaver for convolutional turbo code

For the optimizations applied to the convolutional turbo decoder, we have only regarded the basic block of the decoder: the MAP-module. However, before implementing the turbo decoder on a chip, the interleaver of the decoder also has to be examined.



• Figure 7.1: interleaver scheme

Since the memories within different workers are in conflict with each other (Section 4.4.1), the most straightforward solutions seems to be the splitting of the interleaver into the number of workers that are used for parallelization. This principle is shown in Figure 7.1. However, output bits from different workers (for example from worker B and worker C) can be interleaved into the same worker (worker C) and thus a conflict has been introduced. Currently, a strategy in order to avoid these conflicts is being examined by the turbo-coding group at IMEC.

Implementation issues for product turbo codes

The FBA mainly consists of reordering data. Therefore, most of the applied optimizations for the implementation of the product turbo decoder intended to reduce the number of accesses to the arrays or to reduce the size of the arrays. The arrays were supposed to be implemented as memories. However, a more efficient implementation using registers has been given in [Lit. 18]. Some more possibilities for optimization of the implementation are given below.

One optimization possibility can be found by optimizing the matrix systemization. Different techniques for matrix systemization can be applied (for example the Householder transformation [Lit. 24]).

Another optimization is to pipeline the different functions of the FBA algorithm in order to increase the throughput of the decoder.

DTSE tools

For applying the DTSE-methodology, some tools are being developed at IMEC. Three important tools are the ATOMIUM tool, the Storage Cycle Budget Distribution (SCBD) tool and the Memory Assignment and Allocation (MAA) tool. ATOMIUM is a tool that analyzes the memory usage of array data in C programs and identifies the memory-related bottlenecks by means of C code instrumentation and simulation. The SCBD tool, using a C-code as input, explores different combinations of arrays, resulting in different conflict graphs (Section 4.4.1). The MAA tool uses the output of the SCBD tool for investigating the optimal memory allocation.

The ATMOMIUM tool is a good tool in order to get a quick estimate for the area- and energy consumption. Therefore, we have used ATOMIUM for estimating the area- and energy consumption of the product turbo decoder. Although the other tools (SCBD and MAA) can be very helpful for certain applications, some aspects need improvement.

The main drawback of the SCBD tool is that in C not all properties of the hardware can be described. The value of a register, for example, is always available at the output of that register. This means that this value can be read for calculation and that the outcome of this calculation can be written in the same register within one clock-cycle. However, the SCBD tool does not consider this yet. Another (practical) problem with the SCBD tool is the fact that a lot of the C-code has to be rewritten and that processing time is quite long (some days!).

The main drawback of the MAA tool is that it does not consider arithmetic operations. Although the DTSE-methodology is a technique for data-dominated applications, the arithmetic operations can have a large impact when some optimizations of the methodology have already been applied.

The SCBD- and MAA tool have not been used in this project because of the reasons mentioned above. However, these comments will be input for further improvement of the tools.

Acronyms

AWGN	Additive White Gaussian Channel
BER	Bit Error Rate
BPSK	Binary Phase Shift Keying
CMOS	Complementary Metal Oxide Semiconductor
DTSE	Data Transfer and Storage Exploration
FBA	Fang Buda Algorithm
FER	Frame Error Rate
FSM	Finite State Machine
IMEC	Interuniversity MicroElectronic Center
LLR	Log-Likelihood Ratio
MAA	Memory Allocation and Assignment
MAP	Maximum A-Posteriori
OSW	Overlapping Sliding Window
PDF	Power Distribution Function
RAM	Random Access Memory
SCBD	Storage Cycle Budget Distribution
SNR	Signal to Noise Ratio
UMTS	Universal Mobile Telecommunication System
VHDL	VHSIC (Very High Speed Integrated Circuits) Hardware Description Language
WLAN	Wireless Local Area Network

Literature

[Lit. 1] C. Shannon, "A mathematical theory of communications", Bell Sys. Tech. Journal, vol 27, October 1948.

[Lit. 2] C. Berrou, A. Glavieux, P. Thitimajshima, "Near Shannon limit error-correcting coding and decoding: turbo-codes", Proc. IEEE ICC, pp.1064-1070, May 1993.

[Lit. 3] C. Berrou, P. Adde, E. Angui, S. Faudeil. "A low complexity soft-output viterbi decoder architecture" Proc. IEEE Icc, pp 1064-1070, May 1993.

[Lit. 4] 3rd Generation Partnership Project (3GPP), Technical Specification Group (TSG), Radio Access Network (RAN), Working Group1, "Multiplexing and channel coding", TS 25.222 V1.0.0 Technical Specification, 1999-04.

[Lit. 5] F. Catthoor, S. Wuytack, E. de Greef, F. Balasa, L. Nachtegale, A. Vandecapelle, "Custom memory management methodology, exploration of memory organisation for embedded multimedia system design", Kluwer Academic Publisher, 1998.

[Lit. 6] M. Bossert, "Kanalcodierung", B.G. Teubner Stuttgard, 1998. 2., vollständig neubearb. Und erw. Aufl.

[Lit. 7] J. Hagenauer, E. Offer, L. Papke, "Iterative decoding of binary block and convolutional codes", IEEE Transactions in Information Theory, vol. IT-42, pp 429-445, March 1996.

[Lit. 8] C. Schurgers, L. van der Perre, M. Engels, H. de Man, " Adaptive turbo decoding for indoor wireless communication", International Symposium on Signals and Systems and Electronics ISSSE'98, pp. 107-111, September-October 1998.

[Lit. 9] L.R. Bahl, J. Cocke, F. Jelinek, J. Raviv, "Optimal decoding of linear codes for minimizing symbol error rate", IEEE Transactions on Information Theory, IT-20, pp 248-287, March 1974.

[Lit. 10] G. Masera, G. Piccinini, M. Ruo roch, M. Zamboni, "VLSI architectures for Turbo codes", IEEE Transactions in VLSI Systems, 7(3):369-378, September 1999.

[Lit. 11] H. Dawid and H. Meyr, "Real-time algorithms and VLSI Architectures for Soft Output MAP Convolutional Decoding", in Sixth IEEE International Symposium on Personal Indoor and Mobile Radio Communications (PIMRC'95). Wireless: Merging onto the Information Superhighway (Cat. No.95TH8135), vol. 3, Toronto, pp. 193-197, September 1995.

[Lit. 12] P. Robertson, E. Villebrun and P. Hoeher, "A Comparison of Optimal and Sub-Optimal MAP Decoding Algorithms Operating in the Log Domain", in IEEE International Conference on Communications, pp. 1009-1013, 1995.

[Lit. 13] C. Schurgers, M. Engels, F. Catthoor, "Energy Efficient Data Transfer and Storage Organization for a MAP Turbo Decoder Module", International Symposium on Low Power Electronics and Design (ISLPED '99), San Diego, California, August 1999.

[Lit. 14] Jochen Giese, "Advanced Coding for Satellite Communication", master-thesis IMEC, Leuven, 1999.

[Lit. 15] Keith Baker, Guido Gronthoud, Maurice Lousberg, Ivo Schanstra and Charles Hawkins, "Defect-Based Delay Testing of Resistive Vias-Contacts – A Critical Evaluation", Int. Test Conf. pp. 467-476, 1999.

[Lit. 16] C. Schurgers, "Evaluation of implementation alternatives for a SISO turbo decoder", IMEC Internal Report, 1999.

[Lit. 17] A. Worm, H. Michel, F. Gilbert, G. Kreiselmaier, M. Thul, N. When, " Advanced Implementation Issues of Turbo-Decoders", 2nd International Symposium on Turbo Codes, September 2000.

[Lit. 18] Skybridge document: "DJD coding and interleaving".

[Lit. 19] The Atomium Club, "Atomium 1.1.4 User's Manual", December 1999.

[Lit. 20] L.R. Bahl, J. Cocke, F. Jenelik, J. Raviv, *"Optimal decoding of linear codes for minimizing symbol error rate"*, IEEE Transactions on Information theory, March 1974.

[Lit. 21] Ramesh M. Pyndiah, *"Near-Optimum Decoding of Product Codes: Block Turbo Codes"*, IEEE Trans. On Communications, vol. 46, no. 8, Aug. 1998

[Lit. 22] A. Giulietti, J. Liu, F. Maessen, and etc., "A trade-off study on concatenated channel coding techniques for high data rate satellite communications", accepted by the 2nd International Symposium on Turbo Codes & Related Topics, Brest, Sep. 2000

[Lit. 23] W. H. Press, S.A. Teukolsky, W.T. Vetterling, B.P. Flannery, "Numerical Recipes in C: the art of scientific computing", Cambridge University Press, 1992.

[Lit. 24] "Householder transformations and Golub's Method", Stanford University.

[Lit. 25] A. Giulietti, M. Sturm, F. Maessen, B. Gyselinckx, L. van der Perre, *"A study on fast, low-power VLSI architectures for turbo codes"*, International Microelectronics Symposium and Packaging, Brazil, Sep. 2000

[Lit. 26] F. Maessen, L. van der Perre, F. Willems, B. Gyselinckx, M. Engels, F. Catthoor, *"Memory power reduction for the high-speed implementation of turbo codes"*, Symposium on Vehicular Communications and Technology, Leuven, Oct. 2000.

[Lit. 27] Pyndiah-R; Picart-A; Glavieux-A, "*Performance of block turbo coded 16-QAM and 64-QAM modulations*", GLOBECOM '95, IEEE Global Telecommunications Conference, IEEE, New York, pp. 1039-1043 vol.2, 1995.

Appendix A Alcatel 0.35-µm CMOS technology

This appendix gives the characteristics for the 0.35-µm CMOS technology of Alcatel microelectronics in terms of energy, area and timing.

A.1. Memories

The following formulas for the area and power- consumption within memories have been derived at IMEC by means of synthesis:

$$A = 10400 + 125W + 2650L + 32WL\left[\mu m^2\right] \quad (A.1)$$

$$P = 18.5 + 0.0203W + 9.93L + 0.017WL + 6.73\log_2 W\left[\mu W / MHz\right] \quad (A.2)$$

These formulas are no measurement results. In these formulas, *W* stands for the number of words in a memory and *L* for the word-length. However, these formulas do not include the wire loads of the memories. For these loads, an average capacity of 0.2 pF has been estimated by using SYNOPSIS (a tool that simulates hardware for a given VHDL-code). Given a supply voltage of 2.7 V and assuming an activity of typically 30%, the energy consumed by the wire loads is:

$$P = \frac{1}{2} \cdot C \cdot V^2 \cdot 0.3 = \frac{1}{2} \cdot 0.2 \cdot 2.7^2 \cdot 0.3 = 0.2187\left[\mu W / MHz\right] \quad (A.3)$$

A.2. Registers

For the energy consumption in the registers, the wire-loads have been included in the same way as for the memories. An average capacity of 0.04 pF has been estimated for the loads of the registers, which results in 0.04374 µW/MHz energy consumption.

$$A = 378.0\left[\mu m^2\right] \quad (A.4)$$

$$P = 0.6355 + 0.04374\left[\mu W / MHz\right] \quad (A.5)$$

A.3. Arithmetic units

The costs of the arithmetic operations in terms of energy and area have been approximated by using SYNOPSIS. The costs in terms of area are directly given by SYNOPSIS. To get an estimation of the energy consumption, an activity of 30% is typically assumed. The energy- and area estimations have been based on units of 8 bits. The energy consumption indicates the energy that is needed for one operation.

Arithmetic unit	Energy (pJ)	Area (µm^2)
Adder	10.14	7447
Subtractor	11.91	9594
Comparator	4.068	3851
Max* operator	24.86	20809
Counter	3.49	3708

A.4. Length of the critical path

Simulations with SYNOPSIS have resulted in the following timing-schedule for metric calculations:

Des/Clust/Port	Wire Load Model	Library
alpha_calc	3000to6000	MTC45000_WL_TYP

Point	Incr	Path
clock clk (rise edge)	0.00	0.00
clock network delay (ideal)	0.00	0.00
intrinsic_at1_reg[2]/CP (FD2SQM)	0.00	0.00 r
intrinsic_at1_reg[2]/Q (FD2SQM)	0.78	0.78 r
add_137/plus/plus/U36/Z (BF2T16)	0.35	1.13 r
U1226/Z (NR2P)	0.19	1.32 f
add_137/plus/plus/U64/Z (NR2P)	0.21	1.53 r
U1181/Z (ND3P)	0.18	1.71 f
add_137/plus/plus/U70/Z (ND4P)	0.26	1.97 r
add_137/plus/plus/U48/Z (ENX3)	0.81	2.79 r
U1188/Z (OR2AX4)	0.47	3.26 r
add_138/plus/plus/U46/Z (AO6)	0.26	3.53 f
add_138/plus/plus/U79/Z (AO7P)	0.35	3.87 r
add_138/plus/plus/U63/Z (EN)	0.57	4.44 f
add_138/plus/plus/U83/Z (BF2T16)	0.43	4.87 f
add_149/plus/plus/U38/Z (AN2X4)	0.43	5.30 f
U1190/Z (AO6P)	0.34	5.64 r
U1191/Z (AO7P)	0.23	5.86 f
add_149/plus/plus/U63/Z (AO6P)	0.31	6.18 r
U720/Z (ENX3)	0.80	6.98 r
U1187/Z (NR2AP)	0.18	7.16 f
calc_min_129/sub_59/minus/minus/U74/Z	0.33	7.49 r
calc_min_129/sub_59/minus/minus/U77/Z	0.17	7.67 f
calc_min_129/sub_59/minus/minus/U90/Z	0.46	8.13 r
calc_min_129/sub_59/minus/minus/U76/Z	0.63	8.75 f
calc_min_129/sub_59/minus/minus/U93/Z	0.43	9.18 f
U513/Z (MUX21P)	0.74	9.92 r
U1204/Z (NR2P)	0.22	10.15 f
calc_min_129/gt_66/gt/gt/U11/Z (AN2X3)	0.39	10.53 f
calc_min_129/gt_66/gt/gt/U8/Z (AO1P)	0.48	11.02 r
U569/Z (NR2P)	0.30	11.32 f
U1178/Z (OR2X3)	0.45	11.78 f
U1171/Z (AN3AX3)	0.41	12.19 f
r829/U47/Z (ND2P)	0.20	12.39 r
r829/U44/Z (EN)	0.45	12.84 f
alpha_tmp_reg[3][8]/D (FD2SM)	0.00	12.84 f
data arrival time		**12.84**

Appendix B Mathematical derivations in the MAP-algorithm

This appendix gives the mathematical derivation for the Log-Likelihood Ratio within the MAP-algorithm and the functions of the *LLR*, the extrinsic information and the metric calculation in the logarithmic domain.

B.1. Derivation of the *LLR*

The *LLR* is the output of the turbo decoder. It is defined as the logarithm of the possibility that the transmitted bit is a *one* divided by the possibility that transmitted bit is a *zero*. The derivation of the *LLR* is given below.

$$LLR\left(c_t^s\right) = \log \frac{P_{a-posteriori}\left(c_t^s = 1\right)}{P_{a-posteriori}\left(c_t^s = 0\right)}$$

= (substitute with equation 2.9 and equation 2.10)

$$\log \frac{\sum_{(s',s)\in B_1} \alpha_{t-1}(s')\cdot P_{metric-transition}(s',s)\cdot \beta_t(s)}{\sum_{(s',s)\in B_0} \alpha_{t-1}(s')\cdot P_{metric-transition}(s',s)\cdot \beta_t(s)}$$

= (substitute $P_{metric\text{-}transition}$ with equation 2.12)

$$\log \frac{\sum_{(s',s)\in B_1} \alpha_{t-1}(s')\cdot e^{-\left(\frac{\left|Y_t - a\cdot(2C_t(s',s)-1)\right|^2}{2\sigma^2}\right)}\cdot P\left(c_t^s(s',s)=1\right)\cdot \beta_t(s)}{\sum_{(s',s)\in B_0} \alpha_{t-1}(s')\cdot e^{-\left(\frac{\left|Y_t - a\cdot(2C_t(s',s)-1)\right|^2}{2\sigma^2}\right)}\cdot P\left(c_t^s(s',s)=0\right)\cdot \beta_t(s)}$$

= (in the exponent, only the in-product of Y and C is different in nominator and denominator:
- Y does not depend on the state-transition
- the quadruple of $(2C-1)$ is always 1)

$$\log \frac{\sum_{(s',s)\in B_1} \alpha_{t-1}(s')\cdot e^{-\left(-\frac{2aY_t\cdot(2C_t(s',s)-1)}{2\sigma^2}\right)}\cdot P\left(c_t^s(s',s)=1\right)\cdot \beta_t(s)}{\sum_{(s',s)\in B_0} \alpha_{t-1}(s')\cdot e^{-\left(-\frac{2aY_t\cdot(2C_t(s',s)-1)}{2\sigma^2}\right)}\cdot P\left(c_t^s(s',s)=0\right)\cdot \beta_t(s)}$$

= (write out the in-product in the exponent)

$$\log \frac{\sum_{(s',s)\in B_1} \alpha_{t-1}(s') \cdot e^{\left(\frac{4a\left(c_t^s(s',s)y_t^s + c_t^{p1}(s',s)y_t^{p1} + c_t^{p2}(s',s)y_t^{p2}\right) - 2a\left(y_t^s + y_t^{p1} + y_t^{p2}\right)}{2\sigma^2}\right)} \cdot P\left(c_t^s(s',s)=1\right) \cdot \beta_t(s)}{\sum_{(s',s)\in B_0} \alpha_{t-1}(s') \cdot e^{\left(\frac{4a\left(c_t^s(s',s)y_t^s + c_t^{p1}(s',s)y_t^{p1} + c_t^{p2}(s',s)y_t^{p2}\right) - 2a\left(y_t^s + y_t^{p1} + y_t^{p2}\right)}{2\sigma^2}\right)} \cdot P\left(c_t^s(s',s)=0\right) \cdot \beta_t(s)}$$

= (c_t^s is *zero* in the denominator and *one* in the nominator)

$$\log \frac{\sum_{(s',s)\in B_1} \alpha_{t-1}(s') \cdot e^{\left(\frac{\left(4a\left(y_t^s + c_t^{p1}(s',s)y_t^{p1} + c_t^{p2}(s',s)y_t^{p2}\right) - 2a\left(y_t^s + y_t^{p1} + y_t^{p2}\right)\right)}{2\sigma^2}\right)} \cdot P\left(c_t^s(s',s)=1\right) \cdot \beta_t(s)}{\sum_{(s',s)\in B_0} \alpha_{t-1}(s') \cdot e^{\left(\frac{\left(4a\left(c_t^{p1}(s',s)y_t^{p1} + c_t^{p2}(s',s)y_t^{p2}\right) - 2a\left(y_t^s + y_t^{p1} + y_t^{p2}\right)\right)}{2\sigma^2}\right)} \cdot P\left(c_t^s(s',s)=0\right) \cdot \beta_t(s)}$$

= (rewrite the formula)

$$\frac{2a}{\sigma^2} \cdot y_t^s + \log \frac{P\left(c_t^s(s',s)=1\right)}{P\left(c_t^s(s',s)=0\right)} + \log \frac{\sum_{(s',s)\in B_1} \alpha_{t-1}(s') \cdot e^{\left(\frac{4a\left(c_t^{p1}(s',s)y_t^{p1} + c_t^{p2}(s',s)y_t^{p2}\right)}{2\sigma^2}\right)} \cdot \beta_t(s)}{\sum_{(s',s)\in B_0} \alpha_{t-1}(s') \cdot e^{\left(\frac{4a\left(c_t^{p1}(s',s)y_t^{p1} + c_t^{p2}(s',s)y_t^{p2}\right)}{2\sigma^2}\right)} \cdot \beta_t(s)}$$

= (rewrite the formula)

$$\frac{2a}{\sigma^2} \cdot y_t^s + L_{a-priori}\left(c_t^s\right) + L_{extrinsic}\left(c_t^s\right)$$

B.2. Logarithmic notation for *LLR*

The logarithmic notation for the *LLR* can be derived as follows:

$$LLR\left(c_t^s\right) = \log \frac{P_{a-posteriori}\left(c_t^s = 1\right)}{P_{a-posteriori}\left(c_t^s = 0\right)}$$

= (substitute with equations 2.9 and 2.10)

$$\log \frac{\left(\sum_{(s',s)\in B_1} \alpha_{t-1}(s') \cdot P_{metric-transition}(s',s) \cdot \beta_t(s) \right)}{\left(\sum_{(s',s)\in B_0} \alpha_{t-1}(s') \cdot P_{metric-transition}(s',s) \cdot \beta_t(s) \right)}$$

= (log(a/b) = log(a)- log(b))

$$\log\left(\sum_{(s',s)\in B_1} \alpha_{t-1}(s') \cdot P_{metric-transition}(s',s) \cdot \beta_t(s) \right) -$$
$$\log\left(\sum_{(s',s)\in B_0} \alpha_{t-1}(s') \cdot P_{metric-transition}(s',s) \cdot \beta_t(s) \right)$$

= (substitute with equation 2.14)

$$\max_{(s',s)\in B_1}{}^{*}(\log(\alpha_{t-1}(s') \cdot P_{metric-transition}(s',s) \cdot \beta_t(s))) -$$
$$\max_{(s',s)\in B_0}{}^{*}(\log(\alpha_{t-1}(s') \cdot P_{metric-transition}(s',s) \cdot \beta_t(s)))$$

= (log(a*b) = log(a) + log(b))

$$\max_{(s',s)\in B_1}{}^{*}\left(\tilde{\alpha}_{t-1}(s') + \tilde{P}_{metric-transition}(s',s) + \tilde{\beta}_t(s)\right) -$$
$$\max_{(s',s)\in B_0}{}^{*}\left(\tilde{\alpha}_{t-1}(s') + \tilde{P}_{metric-transition}(s',s) + \tilde{\beta}_t(s)\right) \quad \text{(B.1)}$$

B.3. Logarithmic notation for metric

One metric of the forward recursion and one metric from the backward recursion are necessary to calculate one output bit. For the calculations in the logarithmic domain, we see from equation B.1 that we need the logarithmic value of the metrics. The forward- and backward recursion are analogues. The logarithmic notation for the forward recursion (α) is given below.

$$\tilde{\alpha}_t(S_t) = \log\left(\sum_{S_{t-1}} \alpha_{t-1}(S_{t-1}) \cdot P_{metric-transition}(S_{t-1}, S_t) \right)$$

= (substitute with equation 2.12)

$$\log\left(\sum_{S_{t-1}} \alpha_{t-1}(S_{t-1}) \cdot e^{-\left(\frac{\left|Y_t - a\cdot(2C_t(s',s)-1)\right|^2}{2\sigma^2} \right)} \cdot P\left(c_t^s(s',s)\right) \right)$$

= (substitute with equation 2.14)

$$\max_{S_{t-1}}^{*}\left(\log\left(\alpha_{t-1}(S_{t-1})\cdot e^{-\left(\frac{|Y_t-a\cdot(2C_t(s',s)-1)|^2}{2\sigma^2}\right)}\cdot P(c_t^s(s',s))\right)\right)$$

= (log(a*b) = log(a) + log(b))

$$\max_{S_{t-1}}^{*}\left(\tilde{\alpha}_{t-1}(S_{t-1})-\frac{|Y_t-a\cdot(2C_t(s',s)-1)|^2}{2\sigma^2}+\tilde{P}(c_t^s(s',s))\right)$$

= (only important is the in-product of Y en C:

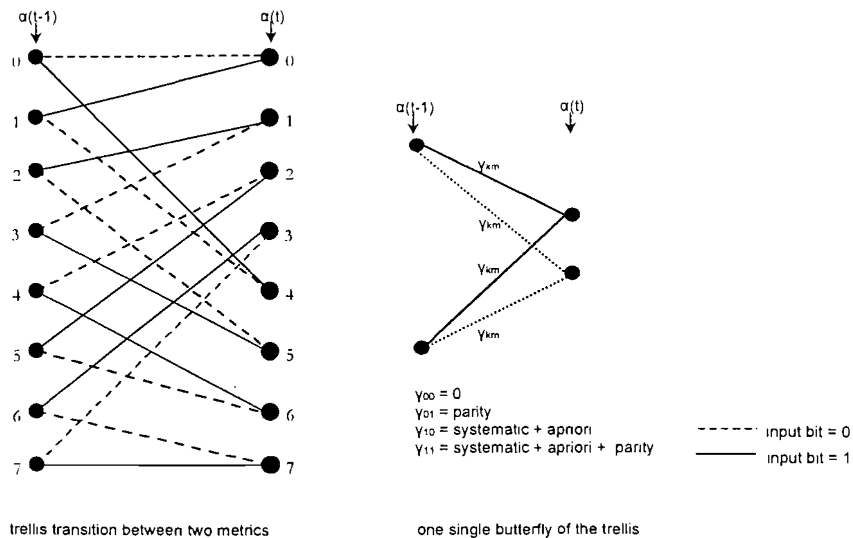
- $(2C-1)^2 = 1$
- Y independent of states -> adds the same value to all states -> output only dependent on difference between states)

$$\max_{S_{t-1}}^{*}\left(\tilde{\alpha}_{t-1}(S_{t-1})+\frac{4a}{2\sigma^2}\cdot Y_t\cdot C_t(s',s)+\tilde{P}(c_t^s(s',s))\right)$$

= (write out in-product)

$$\max_{S_{t-1}}^{*}\left(\tilde{\alpha}_{t-1}(S_{t-1})+\frac{2a}{\sigma^2}\cdot\left(c_t^s(s',s)y_t^s+c_t^{P_1}(s',s)y_t^{P_1}+c_t^{P_2}(s',s)y_t^{P_2}\right)+\tilde{P}(c_t^s(s',s))\right) \quad \text{(B.2)}$$

Equation B.3 shows that the metric is a function of the previous metric and the input values (systematic information, parity information and a-priori information). Which of these values contribute to the value of a certain state within the new metric, depends on the possible transitions within the trellis. This can be depicted with so-called “butterflies”:



- Figure B.1: butterflies to calculate metric

B.4. Logarithmic notation for extrinsic information

The extrinsic information is the soft-output information of one decoder that is sent as a-priori information the next decoder. The derivation of the logarithmic notation for the extrinsic information is given below.

$$extrinsic = \log \frac{\left(\sum_{(s',s) \in B_1} \alpha_{t-1}(s') \cdot e^{parity_info(s,s')} \cdot \beta_t(s) \right)}{\left(\sum_{(s',s) \in B_0} \alpha_{t-1}(s') \cdot e^{parity_info(s,s')} \cdot \beta_t(s) \right)}$$

= (log(a/b) = log(a) – log(b))

$$\log\left(\sum_{(s',s) \in B_1} \alpha_{t-1}(s') \cdot e^{parity_info(s,s')} \cdot \beta_t(s) \right) - \log\left(\sum_{(s',s) \in B_0} \alpha_{t-1}(s') \cdot e^{parity_info(s,s')} \cdot \beta_t(s) \right)$$

= (substitute with equation 2.14)

$$\max_{(s',s) \in B_1}{}^{*} \left(\log\left(\alpha_{t-1}(s') \cdot e^{parity_info(s,s')} \cdot \beta_t(s) \right) \right) - \max_{(s',s) \in B_0}{}^{*} \left(\log\left(\alpha_{t-1}(s') \cdot e^{parity_info(s,s')} \cdot \beta_t(s) \right) \right)$$

= (log(a*b) = log(a) + log(b))

$$\begin{aligned} &\max_{(s',s) \in B_1}{}^{*} \left(\tilde{\alpha}_{t-1}(s') + parity_info(s,s') + \tilde{\beta}_t(s) \right) - \\ &\max_{(s',s) \in B_0}{}^{*} \left(\tilde{\alpha}_{t-1}(s') + parity_info(s,s') + \tilde{\beta}_t(s) \right) \end{aligned} \qquad (B.3)$$

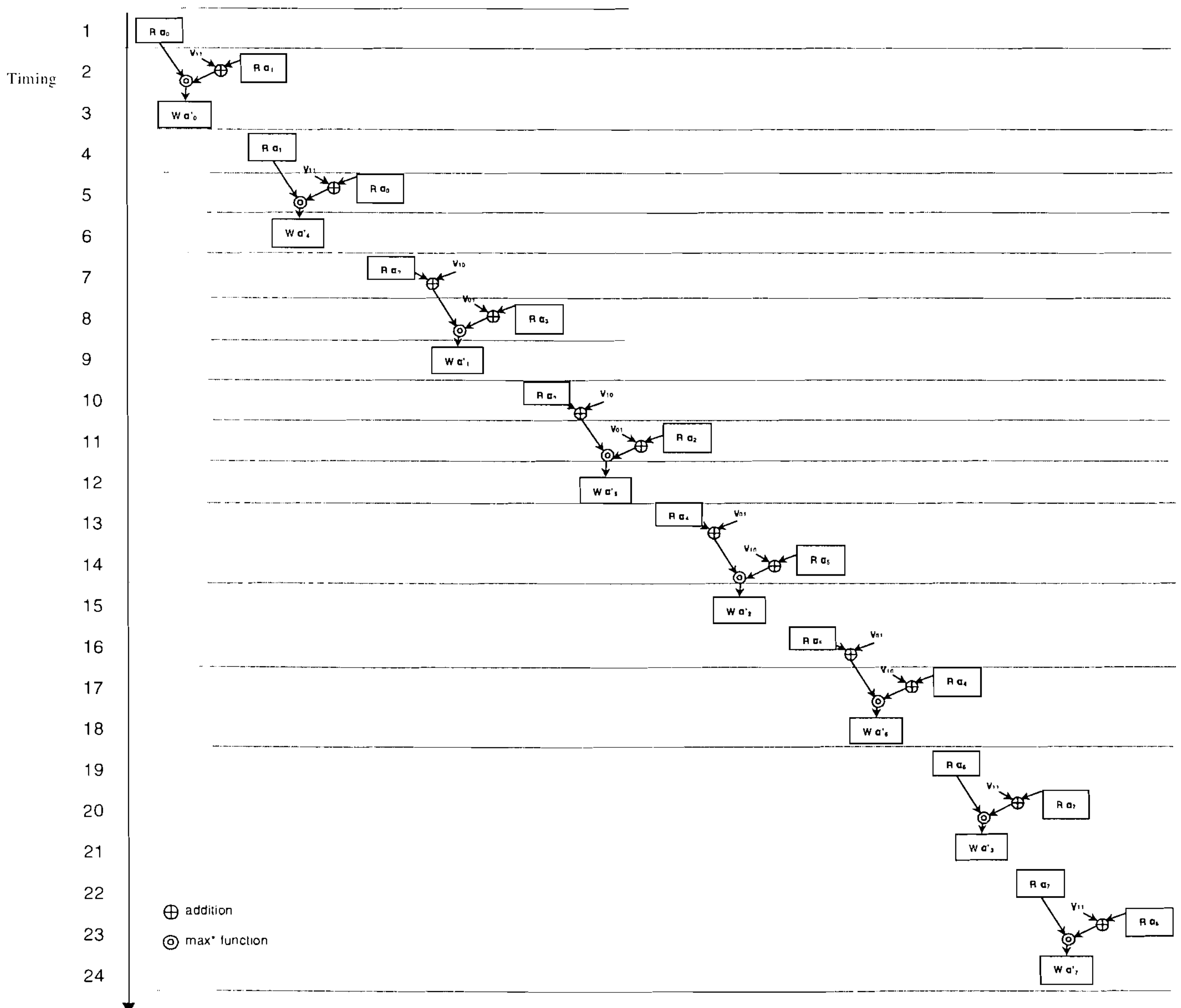
In equation B.3 we can see that the extrinsic information depends on certain combinations of the metrics and the parity information. The combinations of the metrics can be derived from the four butterflies in the trellis (Figure B.1).

Appendix C Implementation of calculations

This appendix gives the implementation of the metric calculation, the subtractive normalization, the channel calculations and the extrinsic calculation. The time-scheduling of the metric calculation and the subtractive normalization is also given because these calculations are in the iterative calculation path.

C.1. Basic implementation metric calculation

The implementation of the forward- and backward recursion is similar. The implementation of equation B.2 in the logarithmic domain is given below. The reads and the writes to the metric memory are depicted with respectively R and W. Since every read- and write operation to the metric memory costs one clock-cycle, we need three clock-cycles for the calculation of each state-metric. Assuming eight states, 24 clock-cycles are needed for the calculation of one metric. The γ values are combinations of input values. The implementation of these combinations is given in the next section.

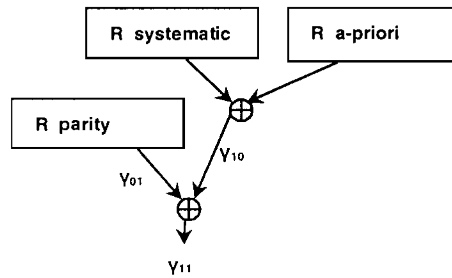


C.2. Basic implementation input calculation

The combination of input values needed to calculate a metric are depicted in as γ:

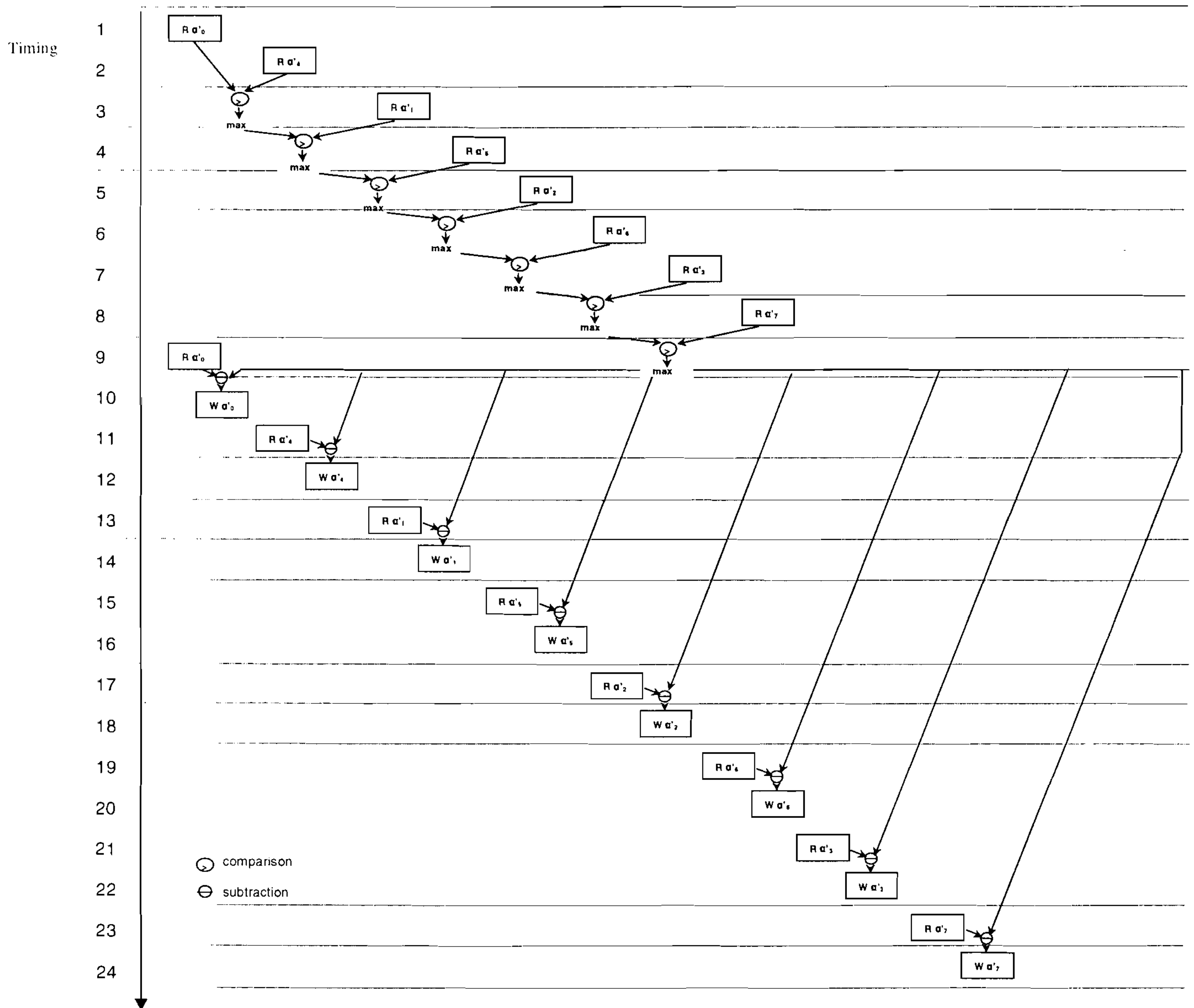
- γ_{01} = parity
- γ_{10} = systematic + a-priori
- γ_{11} =systematic + a-priori + parity

The implementation of the calculation of the input values is given below.



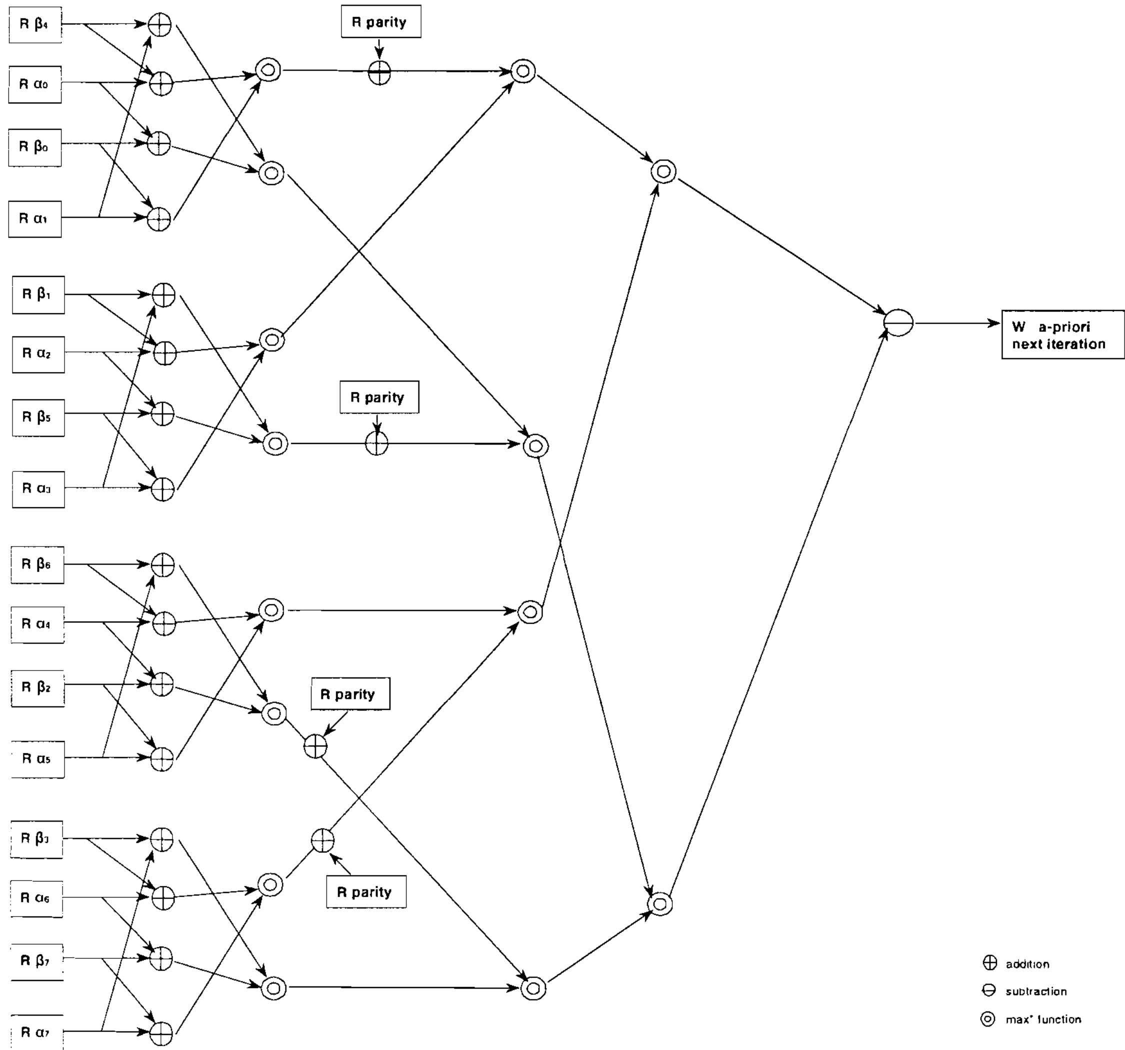
C.3. Basic implementation subtractive normalization

The implementation of the subtractive normalization is given below. Every read- and write to the metric memory costs one clock-cycle, resulting in 24 clock-cycles for the subtractive normalization.



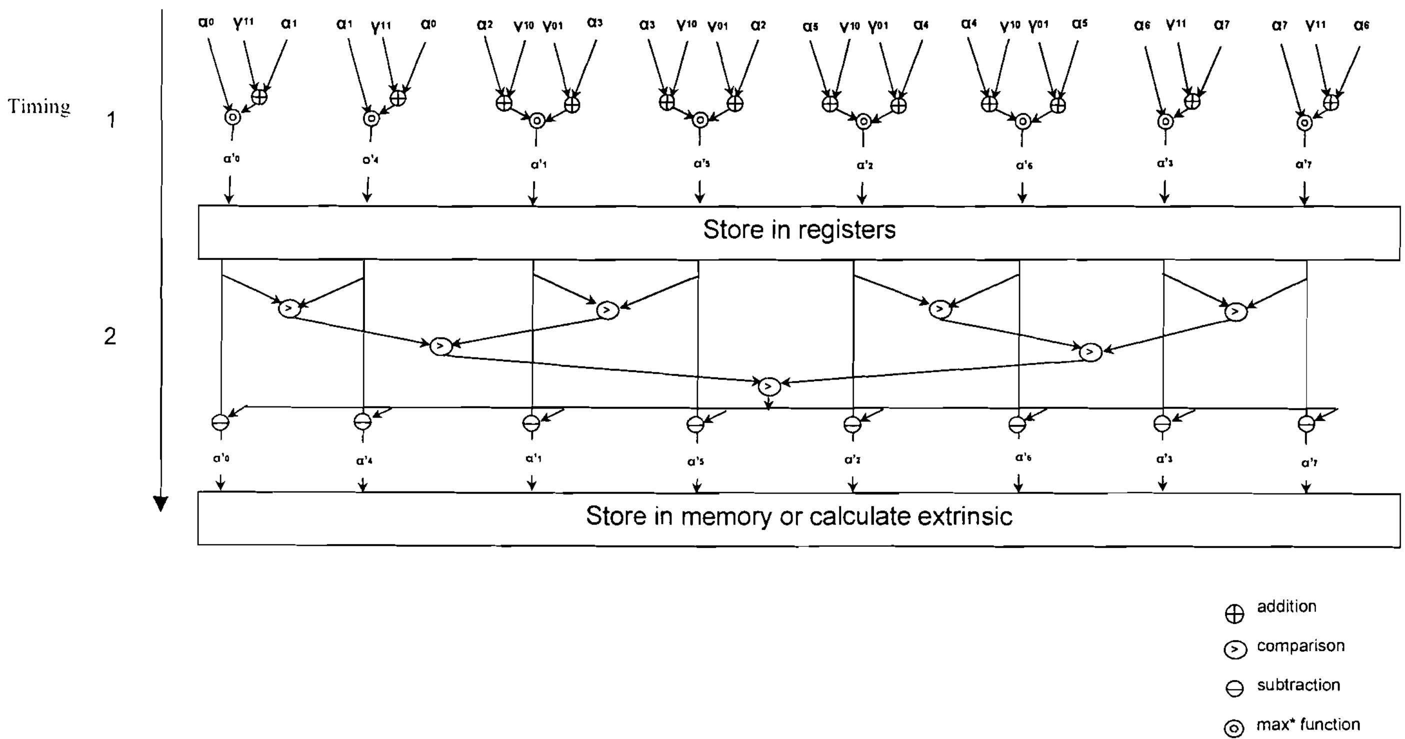
C.4. Basic implementation extrinsic output

The implementation of the extrinsic output (equation B.3) is given below.



C.5. Optimized implementation metric calculation and normalization

The optimized implementation of the metric calculation and subtractive normalization is given below. As we can see, one clock-cycle is needed for the metric calculation and one for the normalization. However, no extra clock-cycle is needed when applying the normalization scheme in Figure 4.10.



Appendix D Computations on the convolutional turbo code

This appendix gives the computations for the estimations of the convolutional turbo code after each optimization step. These numbers were derived from the algorithm. The number of states (*nb_states*) is 8. The frame-length (*frame_length*) is 400 and the number of iterations (*nb_iter*) 6. The number of words in a memory is denoted as *W* and the word-length as *L*.

D.1. Computations after loop transformations

Numbers of calculations after loop transformations:

- Metric-calculations:
  $2 \cdot (frame_size + nb_dummies \cdot nb_workers) \cdot nb_iter \cdot 2$
- Subtractive normalization:
  $2 \cdot (frame_size + nb_dummies \cdot nb_workers) \cdot nb_iter \cdot 2$
- Extrinsic-calculations:
  $frame_size \cdot nb_iter \cdot 2$
- Channel-calculations:
  $2 \cdot nb_states \cdot (frame_size + nb_dummies \cdot nb_workers) \cdot nb_iter \cdot 2$

Numbers of accesses to the memories after loop transformations:

- Metric-memory (W = *worker_size*/2*8 ; L = 7):
  $((2 \cdot 6 \cdot nb_states + 16) \cdot frame_size + 6 \cdot nb_states \cdot nb_workers \cdot nb_dummies) \cdot nb_iter \cdot 2$
- Systematic-input memory (W = *worker_size*/2 ; L = 4):
  $2 \cdot nb_states \cdot (frame_size + nb_workers \cdot nb_dummies) \cdot nb_iter \cdot 2$
- Parity-input memory (W = *worker_size*/2; L = 4):
  $(2 \cdot nb_states \cdot (frame_size + nb_workers \cdot nb_dummies) + 4 \cdot frame_size) \cdot nb_iter \cdot 2$
- A-priori memory (W = *worker_size*/2; L = 6):
  $(2 \cdot nb_states \cdot (frame_size + nb_workers \cdot nb_dummies) + frame_size) \cdot nb_iter \cdot 2$

Number of clock-cycles after loop transformations is *(worker_size+18)*576*:

- 48 clock-cycles are needed to calculate one metric (four memory-reads and two memory-writes to the metric-memory for eight states of the metric).
- One half iteration lasts (worker_size + nb_dummies) metric calculations
- 6 iterations have to be performed for one frame.

D.2. Computations after introducing a memory hierarchy

Numbers of calculations after introducing a memory hierarchy:

- Metric-calculations: $2 \cdot (frame_size + nb_dummies \cdot nb_workers) \cdot nb_iter \cdot 2$
- Subtractive normalization: $2 \cdot (frame_size) \cdot nb_iter \cdot 2$
- Extrinsic-calculations: $frame_size \cdot nb_iter \cdot 2$
- Channel-calculations: $2 \cdot (frame_size + nb_dummies \cdot nb_workers) \cdot nb_iter \cdot 2$

Number of accesses to the memories after introducing a memory hierarchy:

- Metric-memory (W =*worker_size/2**8; L = 7):
  $2 \cdot (nb_states \cdot frame_size) \cdot nb_iter \cdot 2$
- Systematic-input memory (W = *worker_size*/2; L = 4):
  $2 \cdot (frame_size + nb_workers \cdot nb_dummies) \cdot nb_iter \cdot 2$
- Parity-input memory (W = *worker_size*/2; L = 4):
  $2 \cdot (frame_size + nb_workers \cdot nb_dummies) \cdot nb_iter \cdot 2$
- A-priori memory (W = *worker_size*/2 ; L = 6):
  $(2 \cdot (frame_size + nb_workers \cdot nb_dummies) + frame_size) \cdot nb_iter \cdot 2$
- Metric-registers (W = *nb_states**2; L = 10):
  $2 \cdot (frame_size + nb_workers \cdot nb_dummies) \cdot nb_iter \cdot 2$
- Input-registers (W = 2; L = 7); (W = 2; L = 7); (W = 2; L = 4):
  $2 \cdot 3 \cdot (frame_size + nb_workers \cdot nb_dummies) \cdot nb_iter \cdot 2$

Number of clock-cycles after introducing a memory hierarchy is *((worker_size/2)*8+(worker_size/2)+21)*12*:

- 8 clock-cycles are needed to calculate and store one metric (the storage is only needed during worker_size/2)
- 1 clock-cycle is needed for a dummy-calculation and for calculations when the extrinsic is calculated immediately (no storage is needed)
- One half iteration lasts *worker_size* metric calculations, 18 dummy-calculations and 3 calculations for pipelining of the extrinsic calculation
- 6 iterations have to be performed for one frame.

D.3. Computations after memory allocation

Numbers of calculations after memory allocation:

- Metric-calculations: $2 \cdot (frame_size + nb_dummies \cdot nb_workers) \cdot nb_iter \cdot 2$
- Subtractive normalization: $2 \cdot (frame_size) \cdot nb_iter \cdot 2$
- Extrinsic-calculations: $frame_size \cdot nb_iter \cdot 2$
- Channel-calculations: $2 \cdot (frame_size + nb_dummies \cdot nb_workers) \cdot nb_iter \cdot 2$

Number of accesses to the memories after memory allocation:

- Metric-memories (W = *worker_size/2* ; L = 28):
  $2 \cdot (frame_size \cdot 2) \cdot nb_iter \cdot 2$
- Systematic/Parity-input memories (W = *worker_size/2*; L = 8):
  $2 \cdot (frame_size + nb_workers \cdot nb_dummies) \cdot nb_iter \cdot 2$
- A-priori memories (W = *worker_size/2* ; L = 6):
  $(2 \cdot (frame_size + nb_workers \cdot nb_dummies) + frame_size) \cdot nb_iter \cdot 2$
- Metric-registers (W = *nb_states*2*; L = 10):
  $2 \cdot (frame_size + nb_workers \cdot nb_dummies) \cdot nb_iter \cdot 2$
- Input-registers (W = 2; L = 7); (W = 2; L = 7); (W = 2; L = 4):
  $2 \cdot 3 \cdot (frame_size + nb_workers \cdot nb_dummies) \cdot nb_iter \cdot 2$

Number of clock-cycles after memory allocation is *(worker_size+21)*12*:

- 1 clock-cycles are needed to calculate one metric (eight states of the metric can be stored in one clock-cycle)
- 1 clock-cycle is needed for a dummy-calculation (no storage is needed)
- One half iteration lasts *worker_size* metric calculations, 18 dummy-calculations and 3 calculations for pipelining
- 6 iterations have to be performed for one frame.

D.4. Computations after data-flow transformations

Numbers of calculations after data-flow transformations:

- Metric-calculations: $2 \cdot (frame_size + nb_dummies \cdot nb_workers) \cdot nb_iter \cdot 2$
- Subtractive normalization: $2 \cdot (frame_size) \cdot nb_iter \cdot 2$
- Extrinsic-calculations: $frame_size \cdot nb_iter \cdot 2$
- Channel-calculations: $2 \cdot (frame_size + nb_dummies \cdot nb_workers) \cdot 2$

Number of accesses to the memories after data-flow transformations:

- Metric-memories (W = *worker_size/2/theta* ; L = 28):
  $2 \cdot (frame_size \cdot 2 / theta) \cdot nb_iter \cdot 2$
- Systematic/Parity-input memories (W = *worker_size/2*; L = 8):
  $2 \cdot (frame_size + nb_workers \cdot nb_dummies + frame_size - frame_size / theta) \cdot nb_iter \cdot 2$
- A-priori memories (W = *worker_size/2*; L = 6):
  $(2 \cdot (frame_size + nb_workers \cdot nb_dummies) + frame_size + frame_size - frame_size / theta) \cdot nb_iter \cdot 2$

- Cache-memories (W = *theta-1*; L = 28):
  $2 \cdot (frame_size - frame_size / theta) \cdot nb_iter \cdot 2$
- Metric-registers (W = *nb_states**2; L = 10):
  $2 \cdot (frame_size + nb_workers \cdot nb_dummies + frame_size - frame_size / theta) \cdot nb_iter \cdot 2$
- Input-registers (W = 2; L = 7); (W = 2; L = 7); (W = 2; L = 4):
  $2 \cdot 3 \cdot (frame_size + nb_workers \cdot nb_dummies + frame_size - frame_size / theta) \cdot nb_iter \cdot 2$

Appendix E Output from ATOMIUM

This appendix gives the output from ATOMIUM for the non-optimized and the optimized version of the product turbo code. ATOMIUM is a tool that analyzes the memory usage of array data in C programs an identifies the memory related bottlenecks by means of C code instrumentation and simulation

Array	Number accesses non-optimized version	Number accesses optimized version	difference	percentage
Codeword_column	1444344	1366276	-78068	-5%
Codeword_row	1032745	963944	-68801	-7%
Parity_perm_column	1006767	1006773	+6	+0%
Parity_perm_row	511757	511770	+13	+0.003%
Metrics_column	324444	182810	-141634	-4.3%
Metrics_row	334700	177029	-157671	-47%
Perm_column	244168	177268	-66900	-27%
Perm_row	199904	126688	-73216	-36%
Codeword_list_column	192000	192708	+708	+0.3%
Codeword_list_row	155139	154441	-698	-0.45%
Tmp_codewrd_list_column	186088	186989	+901	+0.48%
Tmp_codewrd_list_row	150660	148468	-2192	-1.5%
Codewrd_sgn_column	174850	172826	-2024	-1.2%
Codewrd_sgn_row	181741	177125	-4616	-2.5%
Index_column	66044	65912	-132	-0.2%
Index_row	76779	76409	-370	-0.48%
Hard_dec_perm_column	72912	72912	0	0%
Hard_dec_perm_row	70866	70215	-651	-0.9%
LL_column	54413	53377	-1036	-1.9%
LL_row	47798	46553	-1243	-2.6%
Parity_x	23568	23568	0	0%
Parity_y	44297	44297	0	0%
Cv_column	32627	32529	-98	-0.3%
Cv_row	28573	28195	-378	-1.3%
Metric_codewrd_lst_colmn	11691	9071	-2620	-29%
Metric_codewrd_lst_row	9457	7408	-2049	-21%
V_column	8883	8877	-6	-0.07%
V_row	8762	8699	-62	-0.7%
LLR_column	6944	6944	0	0%
LLR_row	6944	6944	0	0%
Hard_decision_column	6944	6944	0	0%
Hard_decision_row	6944	6944	0	0%
Parity_matrix_column	6404	6404	0	0%
Parity_matrix_row	2303	2303	0	0%
Flag_column	0	10916	+10916	+100%
Flag_row	0	9338	+9338	+100%
TOTAL	6732460	6149874	-582583	-9%

Appendix F Derivation of LLR for product codes

This appendix gives the mathematical derivation for the Log-Likelihood Ratio for the product codes, when assuming BPSK-modulation and an AWGN-channel.

$$LLR = -\log \frac{p(R|L=-1)}{p(R|L=+1)}$$

$$= -\log \frac{\frac{1}{\sqrt{2\cdot\pi}\cdot\sigma}\exp-\left(\frac{(R-(-1))^2}{2\cdot\sigma^2}\right)}{\frac{1}{\sqrt{2\cdot\pi}\cdot\sigma}\exp-\left(\frac{(R-1)^2}{2\cdot\sigma^2}\right)}$$

$$= -\left(-\frac{(R+1)^2}{2\cdot\sigma^2}+\frac{(R-1)^2}{2\cdot\sigma^2}\right)$$

$$= \frac{1}{2\cdot\sigma^2}\left((R+1)^2-(R-1)^2\right)$$

$$= \frac{1}{2\cdot\sigma^2}\cdot 4R$$

Appendix G Sorting algorithms

For the product code, a sorting algorithm is needed. This appendix gives two different types of sorting algorithms in C code.

G.1. Bubble sort

```
void Bubble_Sort (float list[N])
{
/* this algorithm sorts a list by putting the (n+1)th element at the right place between the n
already sorted elements */

int j,k;
float temp;

for (j=0;j<N-1;j++)
 for (k=0;k<N-1-j;k++)
   if (list[k] > list[k+1])
   {
    temp = list[j];
    list[j] = list[k];
    list[k] = temp;
   }
}
```

G.2. Shell's law

```
void Shell (float list[N])
{
/* this algorithm sorts a list of for example 16 numbers n1...n16 by first sorting each of the 8
groups of 2 (n1, n9), (n2, n8), ... ,(n8, n16); then sort each of the 4 groups of 4 (n1, n5, n9, n13), ... ,(n4,
n8, n12, n16); next sort 2 groups of 8 records and finally sort the whole list. */

int j.k, inc;
float temp;

inc = 1;

/* determine the starting increment */
do{
  inc *=3 ;
  inc++
  } while (inc <= N);

/* loop over partial sorts */
do{
  inc /= 3;
  for (j-inc+1;j<=n;j++)
    {
    temp = a[j];
    k = j;
    while (a[k-inc] > temp)
        {
        a[k] = a[k-inc];
        k -= inc;
        if (k <= inc) break;
        }
    a[k] = temp;
    }
  }while (inc > 1);
}
```