VLSI design and implementation of a Turbo Decoder

Dielissen, J.T.M.H.

Award date:
2000

TU/e technische universiteit eindhoven

Faculty of Electrical Engineering
Section Design Technology for Electronic Systems (ICS/ES)

Master's Thesis

VLSI DESIGN AND IMPLEMENTATION OF A TURBO DECODER

J.T.M.H. Dielissen

Coach: ir. J. A. Huisken (Philips Research)
Supervisor: prof.dr.ir. J.L. van Meerbergen
Date: June 2000

VLSI design and implementation of a Turbo Decoder

Student

ing. J.T.M.H. Dielissen
Eindhoven University of Technology
Department Electrical Engineering
Information- and Communication Systems (ICS)
id. no. 452840
j.t.m.h.dielissen@ stud.tue.nl

Supervisors

Prof. Ir. J. L. van Meerbergen
Eindhoven University of technology
Department Electrical Engineering
Information- and Communication Systems (ICS)
Philips Research Laboratories Eindhoven
Embedded Systems Architectures on Silicon (ESAS)
jef.van.meerbergen@philips.com

Ir. J. A. Huisken
Philips Research Laboratories Eindhoven
Embedded Systems Architectures on Silicon (ESAS)
jos.huisken@philips.com

Preface

In the course of my study Electrical engineering at Eindhoven University of Technology, I conducted a research to the implementation and cost of a turbo decoder. The research is conducted as my graduation assignment at Philips Nat.Lab. This report contains the results of the research.

The reason I chose for this assignment was the fact that the assignment contains research topics from two areas within my personal interest: information theory, and computer science. This report will start with an theoretical explanation of turbo decoding. My personal contribution lies more on the computer science part of it, explained in the later chapters.

My research resulted in six patent disclosures, and papers have been sent to symposia in Europe and America. Two papers are already accepted and the notification of acceptance of the third paper is scheduled in June. Immediately after my graduation assignment I will, together with some colleagues, conduct the transfer of the turbo decoder implementation to Philips Semiconductors and Philips Consumer Electronics.

I would like to thank some people, because without whom it would not have been possible to conduct my research is this promising way. First of all I would like to thank my daily supervisor at Philips Nat.Lab. Jos Huisken and my professor from Eindhoven University of Technology, Jef van Meerbergen for their contribution and support. Second of all I would like to thank Arie Koppelaar and Frans Willems for their patience and knowledge in explaining information theory in general, and turbo decoding in particular. And last bot not least I thank all my colleagues for their support and the pleasant time I had at Philips Nat.Lab.

John Dielissen, Eindhoven
7th June 2000

Contents

Samenvatting

De introductie van 'turbo codes' door Berrou in 1993 heeft nieuwe deuren geopend voor kanaal coderings theorieën. De uitstekende bit fout kansen van deze codes hebben een grote hoeveelheid toepassingen mogelijk gemaakt. De turbo code zal onder andere gebruikt gaan worden in de 3^{de} generatie mobiele communicatie (3GPP). Hoe kun je een turbo decoder ontwerpen en wat is de ontwerpruimte tussen prestatie en kosten? Deze vraag staat centraal in dit rapport. Om antwoord te krijgen op deze vraag zullen een aantal decoding/implementatie aspecten uitgelegd en geanalyseerd moeten worden.

Ten eerste is een uitleg van turbo decoding vereist. Turbo decoding is een code waarbij gebruik wordt gemaakt van meerdere komponent codes. Het resultaat van de eerste komponent decoder wordt gebruikt als extra ingang voor het decoderen van de tweede komponent code. Als op deze wijze alle codes zijn gedecodeerd, wordt het resultaat van de allerlaatste komponent decoder weer gebruikt als ingang voor het decoderen van de allereerste komponent code. Vervolgens gaat het decoderen op dezelfde wijze verder tot een bepaald stoppunt. Dit iteratief decoderen levert een zeer hoge prestatie op.

De tweede groep aspecten die geanalyseerd word, zijn de verschillende implementatie aspecten van een turbo decoder. Hierbij moet onder andere gekeken worden naar kwantisatie, logaritmische correctie, schaling van informatie, normalisatie, implementatie technieken en technieken die ervoor zorgen dat de iteraties stoppen als een correcte uitkomst wordt verwacht.

Voor het verkrijgen van concrete getallen voor de verschillende mogelijke implementatie keuzes en voor het vinden van antwoorden op de vraag hoe je een turbo decoder kunt implementeren, is een oefening gedaan. De turbo decoder kan worden geïmplementeerd met een drie-lagen-structuur, waarbij verschillende gereedschappen voor iedere laag worden gebruikt. Ten eerste is er een flexibele laag, ten tweede een data flow laag en ten derde een executie laag. De oppervlakte en vermogensdissipatie van de belangrijkste modules van de turbo decoder zijn laag, vergeleken met het totale oppervlakte begroting (1 cm^2, 1 Watt).

Met de resultaten van de implementatie oefening en de analyses van de implementatie aspecten kan een ontwerpruimte worden opgezet. Deze ontwerpruimte bestaat uit 4 stromen. In iedere stroom kunnen ontwerpbeslissingen worden gemaakt op basis van gepresenteerde resultaten en analyses.

- De eerste stoom is een algoritmische stroom. Indien prestatie en/of flexibiliteit belangrijk zijn, wordt er voor de BCJR algoritme familie gekozen. Deze familie heeft twee varianten. De variantkeuze hangt af van een kosten/prestatie afweging. Naast een keuze voor een familie variant, moet een implementatie variant worden gekozen. Tenslotte wordt er in deze eerste stoom voor een variabele of vaste raamlengte gekozen.
- De tweede stoom is gerelateerd aan het aantal iteraties. Indien het mogelijk is wordt voor een variabel aantal iteraties gekozen. Dit aantal iteraties kan gestuurd worden. Er zijn technieken om de iteraties van de turbo decoder te stoppen als een correcte uitkomst wordt verwacht.
- In de derde stroom wordt kwantisering, zowel op woordbreedte als op vector niveau (normalisatie), besproken. Voor de 3GPP is normalisatie naar een vooraf vastgestelde variable de beste keuze.
- In de vierde en laatste stoom komt het gebruik van meerdere hardware blokken ter sprake. Als de turbo decoder niet snel genoeg is, moet eerst worden geprobeerd meerdere komponent decoders toe te passen. Indien dit vanwege geheugen bandbreedte niet meer kan, kunnen meerdere turbo decoders worden toegepast.

Voor een turbo decoder implementatie is er een ontwerpruimte tussen prestatie en kosten. Deze ontwerpruimte bestaat uit een aantal algoritmische afwegingen en implementatie aspecten. De algorithmen waaruit gekozen kan worden zijn 'modified SOVA' and 'Max(⋆)-log-MAP'. De prestatie van het Max-log-MAP algoritme kan verbeterd worden door middel van schaling van informatie. Ook implementatie aspecten zoals vector compressie hebben invloed op de prestatie. Binnen de kostenruimte zijn er mogelijkheden om te kiezen voor geheugen-intensieve of berekenings-intensieve oplossingen. De turbo decoder kan op klokfrequenties van 100 MHz opereren, waarbij in de komponent decoder iedere klokslag een resultaat berekend word.

Summary

The introduction of the so called 'turbo codes' by Berrou et. al. in 1993 opened up new perspectives in channel coding theory. The outstanding bit error rate performances and the wide range of applications created a large interest in this coding scheme. The first standard which uses turbo coding is the 3^{rd} generation mobile communication (3GPP). How can a turbo decoder be implemented and what is the design space of performance and cost of such a turbo decoder? To get answers to this general question, a number of decoding/implementation aspects have to be explained and analysed.

First of all an explanation of turbo decoding is needed. A turbo code is a code which uses a number of component codes. The output of decoding the first code is used as a-priori information in decoding the second code. If all codes are decoded this way, the output of the last decoder is used as a-priori information for re-decoding the first code. This process continues until some stop criterion is met. The result of this iterative decoding is a very high performance.

The implementation aspects which have to be analysed are: quantization, logarithmic correction, scaling of information, normalisation, implementation techniques, and techniques which make it possible to stop earlier if the output is expected to be correct.

For obtaining cost figures for the different implementation options, and for finding answers on the question on how to implement a turbo decoder, an implementation exercise is done. The turbo decoder can be implemented using three layers of processing. The highest layer of processing is a flexible layer, the middle layer is a data-flow layer, and the lowest layer is the execution layer. Chip area and power dissipation of the most important unit of the turbo decoder are low, compared to the total chip budget (1 cm^2, 1 Watt).

For exploring the design space, results of the implementation exercise and the implementation aspects analysis are used. This design space consists of 4 sub-streams. In each sub-stream implementation decisions can be made, using the implementation results and analysis.

- The first sub-stream is algorithmic. If performance and/or flexibility are important, the BCJR family must be chosen. This family has two implementation variants. The choice for one of them is a performance/cost consideration. Besides the variant choice, an implementation variant has to be chosen. The last topic concerns the window length.
- The second sub-stream is related to the number of iterations. This can either be a constant or a variable number. Technologies are available to stop the iterations if output is expected to be correct.
- In the third sub-stream quantization is considered, both on word width and vector level (normalisation). For 3GPP, normalisation to a predetermined variable is the best choice.
- The fourth and last sub-stream discusses the extension of hardware units. If a turbo decoder is not fast enough, then first of all the number of component decoders is extended. This can be limited by the memory bandwidth. Further speedup can be achieved by extending the number of turbo decoders.

There is design space of performance and cost for a turbo decoder. This design space consists of a number of algorithmic considerations ('modified SOVA' and 'Max($\star$)-log-MAP') and implementation choices. The performance of the Max-log-MAP algorithm can be improved by scaling information. Implementation choices like vector compression also have influence on performance. Within the cost space, there are memory intensive and computation intensive options. The turbo decoder can run at clock frequencies of 100 MHz and higher. The component decoder can compute an output every clock cycle.

Glossary

3GPP	3^{rd} Generation Partnership Project
APP	A-Posteriori Probability
AWGN	Additive White Gaussian Noise
BCJR	algorithm, named after its authors [1]
cNNI	compressed Next Iteration Initialisations
BER	Bit Error Rate
EI-MAP	Efficient Implementation MAP
FDD	Frequency Division Duplex
fifo	first in first out memory
lifo	last in first out memory
LLR	Log Likelihood Ratio
log-MAP	implementation variant of BCJR
lsb	least significant bit (in a word)
MAP	Maximum A-posteriori Probability
Max-log-MAP	implementation variant of BCJR
Max$\star$-log-MAP	implementation variant of BCJR (pronounced as max-star-log-map)
NII	Next Iteration Initialisation
msb	most significant bit (in a word)
PCCC	Parallel Concatenated Convolutional Code
PIL	Prime Interleaver
RSC	Recursive Systematic Coder
SCCC	Serial Concatenated Convolutional Code
SISO	Soft-Input Soft-Output
SOVA	Soft-Output Viterbi-Algorithm
SNR	Signal to Noise Ratio
stack	last in first out memory
stake	state metric vector at specific trellis steps
TDD	Time Division Duplex
trellis	state diagram of RSC as a function of the time
trellis step	transition from one state to the next in a trellis
UMTS	Universal Mobile Telecommunications System
VLIW	Very Long Instruction Word
WER	Word Error Rate

Chapter 1

Introduction

The union of two different fields of study, combined with state of the art technology can lead to a very interesting and complex area of research. In this thesis the worlds of information theory and computer science are united to find solutions on how to implement a turbo decoder. The wide range of applications in general, and the inclusion of turbo decoding in the 3^{rd} generation mobile communication standard in particular, urged the need for answers to the following question:

> *How can a turbo decoder be implemented and what is the design space of performance and cost of such a turbo decoder?*

Where performance is an 'information theory' concept, describing the error rate of the received message, cost are 'computer science' concepts like power dissipation, chip area, timing, and bandwidth. The following sub-questions can be stated:

1. How can a turbo decoder be implemented?
2. What is the design space of performance and cost of the different algorithms?
3. What is the design space of the implementations alternatives?

It is the aim of this thesis to show the implementation of a turbo decoder and to explore the design space between performance and cost. To achieve this aim: algorithms, implementation aspects, and implementation techniques are explained and their performance and cost are analysed. These results are used in the design and implementation exercise of key elements of the turbo decoder. Both the results of the analysis and the implementation of the Soft Input/Soft Output (SISO) unit are used as an input for the design space exploration. Although the implementation aspects and the design space are tuned towards the 3^{rd} generation mobile communication standard, they are more widely applicable and can be used for turbo decoding in general.

This thesis starts in the information theory world with an explanation of turbo decoding in general (chapter 2). During this explanation it will become clear that the most important unit in a turbo decoder is the SISO unit. The different algorithms for this SISO unit are explained in chapter 3. This chapter combines information theory with computer science by showing the cost of algorithms. A choice for the BCJR family algorithms is made. In chapter 4 the implementation aspects of the turbo decoder are explained. Although this chapter has a strong computer science background, information theory is used to understand the bit/word error rate performances of the different implementation techniques, and to exploit algorithmic freedom. To explore and analyse the cost space of turbo decoding, the design and implementation of the turbo decoder is explained in chapter 5. In chapter 6 the design space of the turbo decoder is explored. This chapter also gives the results of this thesis.

Chapter 2

Turbo coding in general

The introduction of the so called "turbo codes" by Berrou et. al. [3] in 1993 opened up new perspectives in channel coding theory. The outstanding BER performance and the wide range of possible applications of this coding scheme created a large interest in turbo codes.

When digital data are transmitted over a noisy channel, there is always a chance that the received data contains errors [13]. The user generally establishes an error rate above which the received data are not usable. If the received data do not meet the error rate requirement, error correction coding can often be used to reduce errors to a level at which they can be tolerated.

The utility of coding was demonstrated by the work of Shannon [20]. In 1948 he proved that if the data source rate is less than a quantity called the channel capacity, communication over a noisy channel with an error probability as small as desired is possible with proper encoding and decoding. It soon became clear that the *real* limit on communication rate was set not by channel capacity but by the *cost* of implementation of coding schemes. The increasing practicality of coding is due to new developments within the field of error-correcting decoding and the dramatic reduction in cost and size of solid state devices.

Error-correction coding is essentially a signal processing technique that is used to improve the reliability of communication on digital channels. Coded digital messages always contain extra or redundant symbols ($M > N$). These symbols are used to accentuate the uniqueness of each message. They are always chosen in such a way to make it very unlikely that the channel disturbance will corrupt enough of the symbols in a message to destroy its uniqueness. Figure 2.1 shows the model of channel coding that is used here.

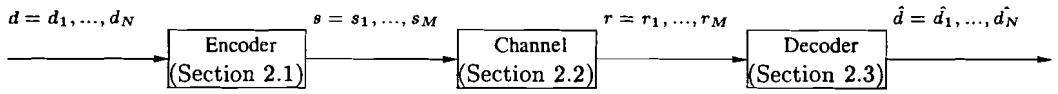

Figure 2.1: Channel coding

The encoder transforms the message (N bits) to M signals $d \rightarrow s$. In the channel, noise is added to the signals. The decoder transforms the M signals received to an estimation of the message (N bits) $r \rightarrow \hat{d}$.

Channel coding theory investigates error correction schemes and codes for minimising the error rates of digital transmissions. Turbo coding is a new form of channel coding and is known for its outstanding Bit Error Rate (average bit error).

Sections 2.1, 2.2, and 2.3 explain turbo coding using the 3 elements of channel coding, described above.

2.1 Turbo encoding

The aim of a channel encoder is to map the incoming message (a word) to a code-word, which is to be sent over the channel. There are 2^N possible messages. The code consists of 2^M words. For channel coding M is always larger than N, implying that not all words in the code are used. Those words which are used are the code-words of the code. By making the Euclidean distance between the code-words as large as possible, we minimise the probability that an incorrect code-word is received.

The turbo code of the 3^{rd} Generation Partnership Project (3GPP) standard contains two component codes. Each component code is encoded with a Recursive Systematic Coder (RSC). The RSC is a part of the structure of a turbo encoder shown in Figure 2.2. The RSC contains a delay line (D) and modulo 2 adders (circled additions). According to the 'feedback polynomial', a parity bit for each input bit is calculated. The generator polynomial of the 8-state (3 delay elements) RSC is shown in equation 2.1. The two RSCs are separated by an interleaver Π, which shuffles the input stream. For each input bit three output bits (systematic and parity bit of RSC 1 and parity bit of RSC 2) are produced, resulting in a rate $\frac{1}{3}$ code. The systematic output bit of RSC 2 does not have to be sent out, since the systematic information for decoding RSC 2 can be produced by interleaving the systematic information for decoding RSC 1.

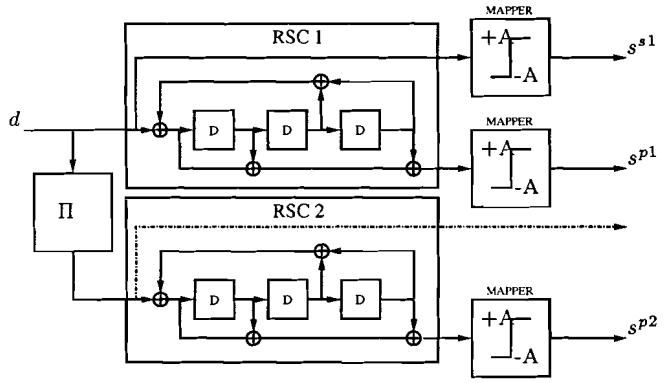

Figure 2.2: 8-state, rate $\frac{1}{3}$ Turbo encoder

Although RSCs produce convolutional codes, this turbo decoding scheme is a block-code. This is due to the interleaver, which divides the input stream into blocks of length N. After processing N input bits the RSCs terminate their trellis resulting in 6 additional bits (3 systematic and 3 parity bits) for each trellis. A trellis is a state diagram of a RSC as a function of the time, as shown in Figure 2.3. The total number of output bits for each block sums up to $M = 3 \cdot N + 12$. While the first part of the encoder uses bits (0,1), the channel assumes continues values $(-\infty \ldots \infty)$. These continues values are further referred to as time discrete signals. The bit value 0 is mapped onto time discrete signal value -A, and the bit value 1 is mapped onto time discrete signal value A. The output of the turbo encoder are M time discrete signals.

$$G(D) = \left[1, \frac{1 + D^1 + D^3}{1 + D^2 + D^3}\right] \tag{2.1}$$

As mentioned before, equation 2.1 shows the generator polynomial of the RSC. This transfer function can be recognised in the structure of the RSC, shown in Figure 2.2. The numerator calculates the parity bit, while the denominator calculates the feedback bit (recursive operation).

The bit values in the delay line elements (D) represent the state of the encoder. The left delay line element represents the most significant bit. Note that there are 2^{μ} states, where μ (=3) is the number of delay elements. Both the next state and the value of the parity bit depends on the current state and the input

bit. The next state is the state in which the RSC ends up after processing the input bit. This can be seen as a Mealy machine. A trellis representation gives a clear view of how the encoder works.

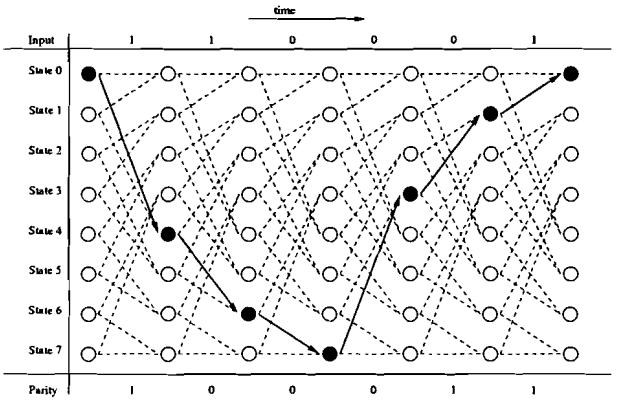

Figure 2.3: Trellis representation

Figure 2.3 shows this trellis. When for example the current state is state 4 and the input bit is 1 then the next state is state 6 and the parity bit is 0. Please note that from any state, any other state can be reached in maximally μ (=3) steps. Returning to state 0 at the end of the trellis is called trellis termination.

2.2 Channel

As explained before, time discrete signals with values -A and +A are sent over the channel. A Bit Error may occur if +A is sent, while the decoder estimates that -A is sent. The channel model we use, is the Additive White Gaussian Noise (AWGN) channel. This channel assumes that the noise is white, Gaussian distributed, and added to the signal. White means that all frequency components are present with equal power in the noise. Figure 2.4 shows the model.

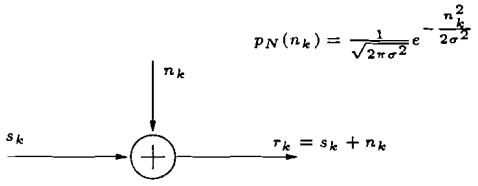

Figure 2.4: Additive Gaussian noise channel

The Gaussian density function is the probability density of the noise values to occur. Equation 2.2 shows the probability density of received symbol r_k, given that the symbol s_k is sent.

$$P_R(r_k|S = s_k) = P_N(r_k - s_k|S = s_k) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(r_k - s_k)^2}{2\sigma^2}} \tag{2.2}$$

Figure 2.5 shows the distribution function for the received signals r, depending on the signal s sent. After multiplying $P(r_k|s_k = A)$ and $P(r_k|s_k = -A)$ with $P(s_k = A)$ and $P(s_k = -A)$ respectively, the decision function is obtained. The value on the x-axis, belonging to the crosspoint of the two distribution functions,is the decision threshold leading to a minimal error probability.

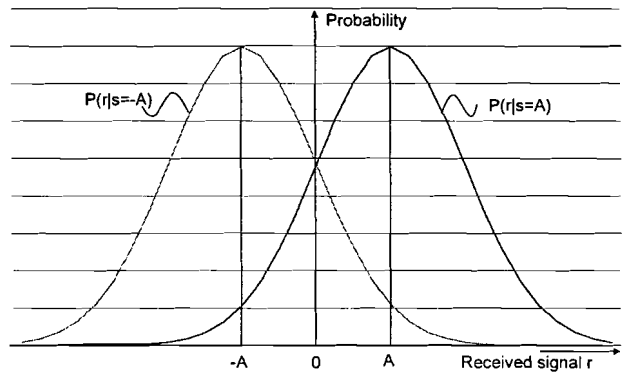

Figure 2.5: Distribution function for received signals r, depending on the signal s sent

A received time discrete signal which is smaller than the threshold d has the highest probability of representing a 0, while a signal higher than the threshold d has a higher probability of representing a 1. The error probability is calculated in equation 2.3. For equal probability signals, the threshold d can be read from figure 2.5 and is equal to 0.

$$\begin{aligned}
Pe &= P(s_k = A) \cdot P(r_k < d|s_k = A) + P(s_k = -A) \cdot P(r_k > d|s_k = -A) \\
&= P(s_k = A) \cdot \int_{-\infty}^{d} P_R(r_k|s_k = A)dr + P(s_k = -A) \cdot \int_{d}^{\infty} P_R(r_k|s_k = -A)dr \\
&\qquad assume: \qquad P(s_k = A) = P(s_k = -A) = \frac{1}{2}, d = 0 \\
&= \int_{0}^{\infty} P_R(r_k|S = -A)dr \\
&= \int_{0}^{\infty} \frac{1}{\sqrt{2\pi\sigma_x^2}} e^{-\frac{(r_k + A)^2}{2\sigma_x^2}} dr \\
&= \int_{A}^{\infty} \frac{1}{\sqrt{2\pi\sigma_x^2}} e^{-\frac{r_k^2}{2\sigma_x^2}} dr \\
&= Q(\frac{A}{\sigma})
\end{aligned} \tag{2.3}$$

The difference between a Maximum Likelihood decision (ML) and a Maximum A-posteriori Probability decision (MAP) lies in the fact that the decision rule for ML is constant, while for MAP the decision rule can change due to a-priori information.

2.3 Turbo decoding

The M time discrete signals received by the turbo decoder do not necessary map on a code-word. In the channel noise is added to these signals, and (implicitly) to the code-word. It is the aim of the turbo decoder to find out which possible code-word has the smallest Euclidian distance to received code r. By finding this code-word the turbo decoder minimises the probability that the message decoded $\hat{d}$ is not equal to the message sent d. Turbo decoding is a special form of decoding because it is working iteratively.

The structure of a Parallel Concatenated Convolution Coder (PCCC) turbo decoder is shown in Figure 2.6. First the Soft Input/Soft Output (SISO) decoder is executed for decoding RSC 1 (Figure 2.2). Then SISO decoder 2 is executed for decoding RSC 2, using extrinsic information λe^{s1} from the first SISO decoder as a-priori information. In the next iteration (a combination of SISO 1 followed by SISO 2 is called an iteration) SISO decoder 1 is executed again, but now using extrinsic information λe^{s2} from the second SISO decoder as a-priori information. This schedule can be continued until some stopping criterion is met.

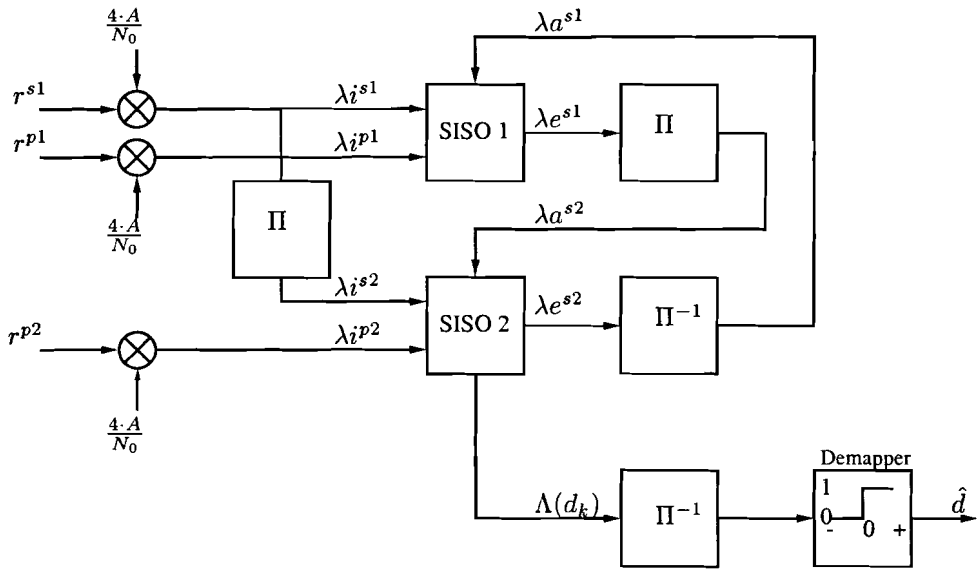

Figure 2.6: Turbo decoder

Multiplication of the received signal r with $\frac{4 \cdot A}{N_0}$ results in intrinsic Log-Likelihood Ratios (LLR) λi. The output of the SISO modules are extrinsic LLRs (λe). The (de-)interleaver transforms the extrinsic LLR λe to a-priori LLR λa for the next SISO decoder. The superscripts stand for: {$\underline{s}$ystematic, $\underline{p}$arity},{ SISO$\underline{1}$, SISO$\underline{2}$}. 'N_0' represents the noise on the AWGN channel ($\sigma^2 = \frac{N_0}{2}$, equation 2.2), and 'A' is the amplitude of the signal transmitted. This transformation to LLR is proven in equation 3.2 on page 26. The SISO decoder calculates extrinsic systematic information (λe^s) and soft-output information ($\Lambda(d_k)$) for each systematic bit received. The relation between extrinsic - and soft-output is explained in equation 3.12 on page 27.

There are four components in this turbo coding scheme that determine the Bit Error Rate (BER) of the output message.

- the Signal to Noise Ratio (SNR)
- the type of interleaving and its interleaving length
- the number of iterations
- the performance of the SISO module

The SNR is an external factor, which influences the BER. The interleaver for a variable interleaving length is given in the 3GPP standard. The number of iterations is a variable that can be changed during execution. For the SISO module two families of SISO algorithms are available: a forward algorithm named Soft Output Viterbi Algorithm [2] (SOVA) and a forward/backward algorithm named after its authors Bahl, Cocke, Jelinek, and Raviv [1] (BCJR). Practical implementation of these algorithms will further increase the BER of turbo decoding. Chapter 3 explains the working and performance of the two algorithm families.

Chapter 3

SISO decoding algorithms

The SISO module is the most important module of turbo coding. At this moment there are two basic algorithm families. As can be found in the literature ([3], [14], and [19]) a large number of algorithms exist in these two families. The two most interesting algorithms for turbo decoding are modified SOVA and Max($\star$)-log-MAP.

Modified SOVA: an algorithm equivalent to Max-log-MAP algorithm, hence not MAP, derived from the Viterbi algorithm family

Max($\star$)-log-MAP: a MAP algorithm on symbols, derived from the BCJR algorithm family

This chapter explains both algorithms by making the basic algorithms intuitively justifiable. This is done to simplify the learning process of the algorithms. For the BCJR algorithm family a theoretical explanation of the derived algorithms is given. For the Viterbi algorithm family no theoretical explanation of the derived algorithm is given, due to the absence of a theoretical background of the derived algorithm (except for the Max-log-MAP equivalence) and the fact that it is not further used in this report.

After explaining the algorithms in Section 3.1 and 3.2, Section 3.3 compares the two most interesting algorithms. This section will also give a justification of the focus on the BCJR family algorithms in this report. In most references the BCJR algorithm is known as the MAP algorithm. In this report this confusing name will be avoided and only the derived algorithms, log-MAP, Max$\star$-log-MAP and Max-log-MAP, will carry that name in it. All examples use the 8-state RSC with the transfer function from Equation 2.1 on Page 15.

3.1 SOVA

The Viterbi algorithm is an efficient hard decision algorithm for implementing trellis decoding. Due to its relatively small computation cost it became very popular. The Viterbi algorithm uses soft input information to produce a hard decision. When using it as a hard decision output algorithm it is a form of optimally decoding the likelihood of a continuous stream of symbols. Alternatives like BCJR (section 3.2) have similar performance for low BERs. On the other hand the SISO module in Turbo decoding has to produce Soft Output. The Viterbi algorithm is modified to produce Soft Output and becomes a Soft Output Viterbi Algorithm (SOVA). Recent developments result in an improved SOVA algorithm. Section 3.1.1 gives a intuitive justification for Viterbi like decoding. Section 3.1.2 evaluates the complexity and performance of the algorithm.

3.1.1 SOVA intuitively justified

The basic problem is how to decode the code produced by the Recursive Systematic Coder. To explain this basic decoding problem first the Viterbi algorithm is explained. This Viterbi algorithm is a hard decision algorithm. The next step in the explanation is the extension of a hard decision to a soft decision Viterbi algorithm, (SOVA).

The first approach is known as the Viterbi algorithm. When starting in state 0 (all delay line elements of the RSC contain bit-value 0) a probability of occurrence for the two possible next states can be calculated. This probability is called the path metric (α_m) of state m. Each state m has 2 outgoing transitions and each transition has a figure for conditional probability of occurrence which we call branch metric ($\gamma_0(\vec{\lambda}_k, m)$ and $\gamma_1(\vec{\lambda}_k, m)$). The relation between the branch metric and the Log Likelihood Ratios (LLR), is stated in equation 3.12 on page 27. If for example the sum of the received Soft input values is positive, the result will be a positive branch metric $\gamma_1(\vec{\lambda}_k, 0)$ from α_0 (0 decimal = 000 binary) to α_4 (4 decimal = 100 binary). The branch metric from α_0 to α_0 ($\gamma_0(\vec{\lambda}_k, 0)$) is always zero due to normalisation. Therefore the next α_4 receives a higher state metric than the next α_0. For the second trellis step, path metrics to state 0, 2, 4, and 6 are calculated, using the path metrics calculated in the first step and the corresponding branch metrics. This repeats for the third step, and now all states have a path metric. When making the fourth step a problem rises. There are two paths leading to each state. At this point a decision is made which path has the highest path metric. The decision is saved and the path metric of the surviving path becomes the maximum path metric of the two paths. This is repeated until the last state is reached. Then the full path can be reconstructed, using the decisions saved during the previous steps. Figure 3.1 shows the algorithm. The arrows represent the decisions, the filled circles are the states visited during the backtracking.

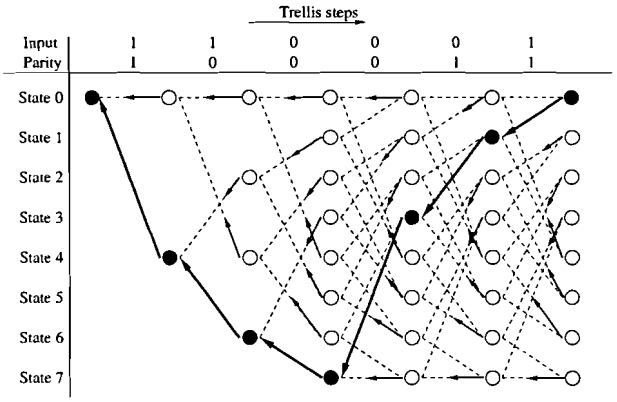

Figure 3.1: Viterbi Algorithm

This is an optimal algorithm for decoding a sequence. The disadvantage of this algorithm is the lack of information about the reliability of each step. When making the decision during every step, information of how close the two paths were together is thrown away. Therefore this algorithm can only be used for making hard decisions. When soft information has to be calculated, as expected from the SISO module, this algorithm can not be used.

For producing soft information not only the decisions are saved, but also the path-metric difference at each place where 2 paths merge is saved. This path-metric difference is used later for producing Soft Output. This algorithm, known as the Soft Output Viterbi Algorithm (SOVA), is shown in Figure 3.2. When reconstructing the most likely path we use the hard decision, starting at the end of the trellis (the solid line). For producing soft information we look at the differences between the most likely path and the most likely path when choosing the other path in first step (dashed line). By calculating the differences between the two paths until they converge, we have a measurement for the accuracy of the decision.

Numerous schemes to calculate soft output can be produced. Modified SOVA is one of the variants. This variant has a strong relation to the Max-log-MAP algorithm of the BCJR family. Explanations of the (modified)-SOVA algorithm can be found in [7], [9], [10], and [18].

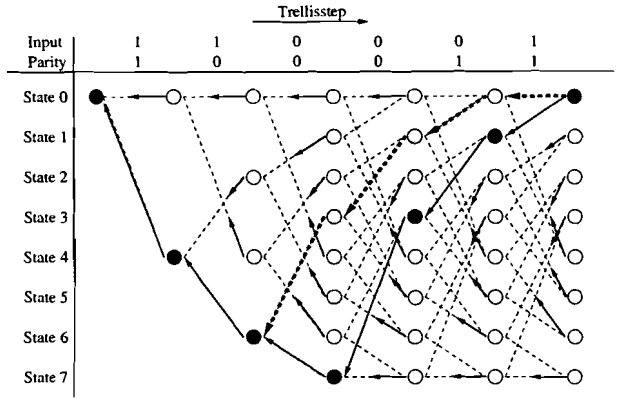

Figure 3.2: Soft Output Viterbi Algorithm

3.1.2 Complexity and performance

Popescu [18] did a complexity analysis of the modified SOVA algorithm. The total resources needed to implement the modified SOVA algorithm according to his report are:

cost i. Operations:
$[3 \cdot 2^{\mu} + 2^{n} + U]$ additions and $[2^{\mu} + 2 \cdot (U-1)]$ compare selects

cost ii. Registers:
$\left[(n+U-1) \cdot P_{\gamma} + 2^{\mu} \cdot (D+U+1) + \frac{U \cdot (U+1)}{2} \cdot (P_{\gamma}+1)\right]$ Registers

cost iii. RAM:
$[D \cdot 2^{\mu} \cdot (P_{\gamma}+1)]$ RAM

where,

- μ = number of delay elements in the RSC, μ=3
- D = survivor depth of the SMU, D= 10 $\cdot(\mu+1) = 40$ in the case of a fading channel
- U = maximum depth of update operation, U = 4$\cdot(\mu+1) = 16$
- n the code rate $\frac{k}{n} = \frac{1}{2}$, i.e. n=2
- P_{γ} = precision, 4 bits

For these figures the resources needed to implemented the modified SOVA algorithm are:

cost i. Operations: 44 Additions, 38 compare selects

cost ii. Registers: 1204

cost iii. RAM: 1600 bit positions

In the literature (Fossorier [7]) it has been shown that Modified SOVA with no finite-length decoding window is theoretically equivalent to Max-log-MAP. The influence of sliding window on the Max-log-MAP algorithm can be compared to finite-length modified SOVA decoding. Simulations results show that the differences between sliding window with training calculations and full window Max-log-MAP become negligible if the training lengths are larger than $10 \cdot (\mu + 1)$ $(= 40)$ in the case of a fading channel. Robertson [19] showed that the difference between Max-Log-MAP (equal to modified SOVA) and Max$\star$-log-MAP is about 0.4dB (N=1024, μ=3 (8-state code), 8 iterations). Simulations conducted at Philips Nat.Lab. and Philips Semiconductors confirm this difference. Due to the lack of correct algorithm descriptions and time, no simulations are done by the author, using the modified SOVA algorithm.

3.2 BCJR

BCJR is an algorithm published in 1974 by Bahl et. al. [1]. The algorithm maximises the a-posteriori probability of a correct decision for each symbol. By origin BCJR is a hard decision algorithm, but when we refer to the BCJR algorithm in Turbo Code, we mean the soft output algorithm. BCJR refers to the first letters of the authors writing the article. The algorithm did not become very popular until recently, because of the required calculation power while it had similar performance for low BERs compared to the Viterbi Algorithm. Turbo code changed this because it requires soft output. In Section 3.2.1 an intuitive justification of the BCJR algorithm is given. Differences between BCJR and SOVA are explained. Section 3.2.2 will give a detailed theoretical explanation of the algorithm. Implementation cost and BER performance are discussed in Section 3.2.3.

3.2.1 BCJR intuitively justified

As in the intuitive justifications for SOVA (section 3.1.1), the hard decision algorithm is explained first. The natural way for decoding the trellis is to reconstruct its trellis in exactly the same way. As mentioned in section 2.1 the result of trellis termination is that the last state is always the zero state. This enables us to not only use the forward recursion of the Viterbi algorithm (decoding from the first state to the last one), but also the other way around: from the last state to the first one (backward recursion). During the recursion, the conditional probabilities of occurrence of the next states are calculated. The probability of occurrence of a state is called the state metric. A state metric is calculated by taking the sum of the previous state and the branch metric of the edge. Since two edges lead to the same state, the maximum state metric is used. For this first approach, the state having the highest state metric is saved.

When decoding a bit, that bit is approached from both the first state (forward recursion) and the last state (backward recursion). The decision is the bit belonging to the transition between the 2 maximum state metrics calculated by the forward and backward recursions respectively. A transition is the edge from one state to the next. Each state has 2 incoming, and 2 outgoing edges. Figure 3.3 shows the decoding of one bit.

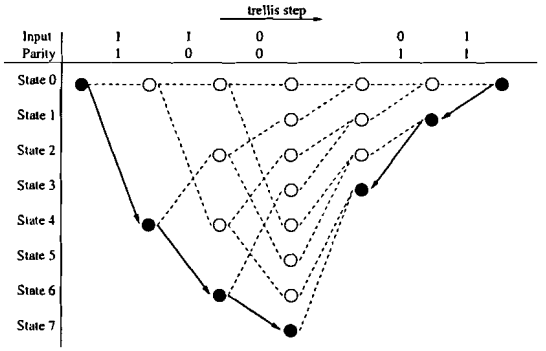

Figure 3.3: Decoding one bit using forward and backward Viterbi

When executing this two way approach for each bit, a possible outline of the algorithm is:

1. reconstruct the trellis in forward direction and save the state number having the highest state metric on a stack (first in last out memory)
2. decode the last bit, using the value on the top of the stack and the known last state
3. reconstruct one trellis step backward (one Viterbi backward step)
4. decode the bit, using the values on the stack top and the reconstruction from step 3
5. jump to step 3 if not ready

Figure 3.4 shows the algorithm displayed in time versus trellis steps.

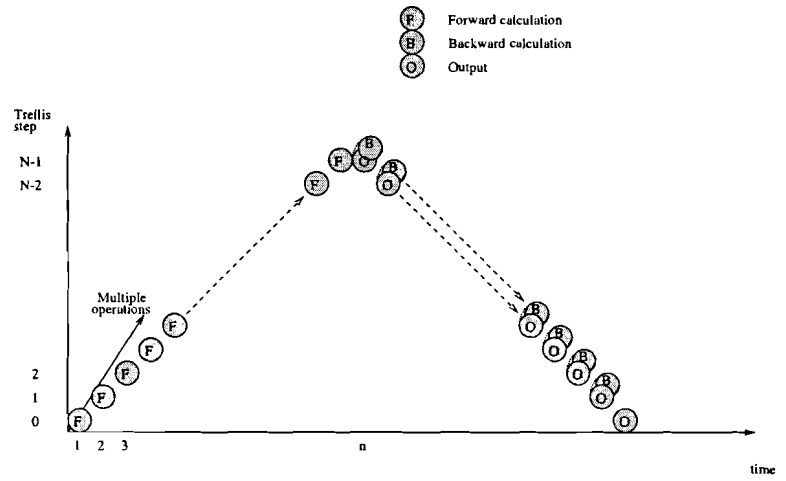

Figure 3.4: BCJR schedule

This algorithm is more expensive than the VA algorithm, since it contains 2 Viterbi algorithms. Problems occur if there is no transition between the maximum state calculated by the forward steps and the maximum state calculated by the backward steps.

To overcome this problem the next solution is used: for making a decision for bit i, the state metrics before i, calculated by the forward recursion, and the state metrics after i, calculated by the backward recursion are used. The transition value between the state before, and the state after bit i is calculated by taking the sum of two state metrics and the branch metric. The bit belonging to the transition with the highest sum of state metrics, is the most likely bit, being transmitted. Figure 3.5 shows the possible transitions. The transition between state 3 (from forward direction) and state 1 (from backward direction) is the largest, resulting in a 0 as output bit.

Note that the left and right states with the highest state metric are not necessarily the states belonging to the transition with the highest transition value.

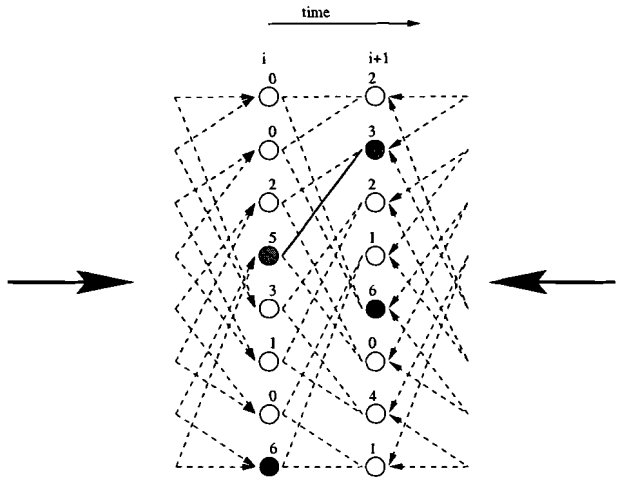

Figure 3.5: BCJR Algorithm, decoding one bit

When using this decoding technique for every bit, a possible outline of the algorithm is:

1. calculate the state metrics for every state for every trellis step in forward direction and save them on a stack
2. find the bit belonging to the transition with the highest transition value, using the values on the top of the stack and the known last state metrics
3. calculate the state metrics one trellis step backwards
4. find the bit belonging to the transition with the highest transition value, using the values on the stack top, the state metrics calculated in step 3, and the branch metrics
5. jump to step 3 if not ready

The layout of this algorithm is exactly the same as the previous algorithm. Again Figure 3.4 shows the algorithm in displayed time against trellis steps. In essence this is the hard decision BCJR algorithm. This algorithm has high memory requirements due to the first algorithm step.

3.2.2 Theory and implementation

When explaining the BCJR algorithm we first have to explain the term soft information. Soft information is the Log-Likelihood Ratio (LLR) of two probabilities and is calculated using equation 3.1

$$\lambda(x) = ln\left(\frac{P(d_k = 1|R = x)}{P(d_k = 0|R = x)}\right) \tag{3.1}$$

There are 2 reasons for using LLRs. The first reason is the easy conversion of received values r_k^s, r_k^{p1}, and r_k^{p2} to this LLR. Equation 3.2 shows the mathematical derivation. Equation 2.2 (Section 2.2 page 17) is used as the probability density function of the noise.

$$
\begin{aligned}
SoftInfo(r_k) &= ln\left(\frac{P(d_k=1|R=r_k)}{P(d_k=0|R=r_k)}\right) \\
&= ln\left(\frac{\frac{P(d_k=1)\cdot P_R(R=r_k|d_k=1)}{P(R=r_k)}}{\frac{P(d_k=0)\cdot P_R(R=r_k|d_k=0)}{P(R=r_k)}}\right), P(d_k=0)=P(d_k=1)=\frac{1}{2} \\
&= ln\left(\frac{P_R(R=r_k|d_k=1)}{P_R(R=r_k|d_k=0)}\right) \\
&= ln\left(\frac{\frac{1}{\sqrt{2\pi\sigma_x^2}}e^{-\frac{(r_k-A)^2}{2\sigma_x^2}}}{\frac{1}{\sqrt{2\pi\sigma_x^2}}e^{-\frac{(r_k+A)^2}{2\sigma_x^2}}}\right) \\
&= \frac{-(r_k-A)^2}{2\sigma_x^2} - \frac{-(r_k+A)^2}{2\sigma_x^2} \\
&= \frac{2\cdot A\cdot r_k}{\sigma_x^2}, \sigma_x^2 = N_0/2 \\
&= \frac{4\cdot A\cdot r_k}{N_0}
\end{aligned} \tag{3.2}
$$

Note that the LLR of a symbol scales linearly with the value of the received sample. This is far easier to calculated than the probability, for which the calculation of exponentials is required. This LLR transformation is included in Figure 2.6 on page 18. The second reason of the use of LLR is the mathematical simplifications possible for the BCJR algorithm. The soft output of the BCJR algorithm (Λ) is calculated using the next equation:

$$
\Lambda(d_k) = ln\frac{\sum_{m'=0}^{2^\mu-1}\alpha_k(m')\cdot\gamma_1(\vec{r}_k,m')\cdot\beta_{k+1}(S_f^1(m'))}{\sum_{m'=0}^{2^\mu-1}\alpha_k(m')\cdot\gamma_0(\vec{r}_k,m')\cdot\beta_{k+1}(S_f^0(m'))} \tag{3.3}
$$

where :

$$
\alpha_k(m) = \sum_{i=0,1}\alpha_{k-1}(S_b^i(m))\cdot\gamma_i(\vec{r}_{k-1},S_b^i(m)) \tag{3.4}
$$

$$
\beta_k(m) = \sum_{i=0,1}\beta_{k+1}(S_f^i(m))\cdot\gamma_i(\vec{r}_k,m) \tag{3.5}
$$

$$
\gamma_i(\vec{r}_k,m) = Pr(r_k^s|d_k=i)\cdot Pr(r_k^p|d_k=i,S_k=m) \tag{3.6}
$$

where additionally:

$\Lambda(d_k)$	soft-output information over bit d_k
$\alpha_k(m)$	forward recursion probability of state m in trellis step k
$\beta_k(m)$	backward recursion probability of state m in trellis step k
$S_f^i(m)$	next state in forward recursion if current state is m and systematic input bit equals i
$S_b^i(m)$	next state in backward recursion if current state is m and systematic input bit equals i
$\gamma_i(\vec{r}_k,m)$	transition probability if current state is m and input bit equals i, using received symbols of trellis step k
$\vec{r}_k$	received symbols for systematic and parity information
μ	number of delay elements in the RSC

Equation 3.3 essentially states the algorithm explained in section 3.2.1. Instead of making a hard decision for each bit, a soft output is calculated. First the sum of all transition values to a positive output bit, is divided by the sum of all transition values to a negative bit. Then the logarithm is taken, leading to a LLR. Note that this algorithm does not use LLRs for Soft Input.

Equations 3.3 to 3.6 show that a hardware implementation is very expensive. This is due to the number of multiplication and the logarithmic function. P. Robertson [19] proposed a number of mathematical simplifications. The first step is to take the Logarithm of alpha, beta and gamma (3.7).

$$\tilde{\alpha}_k(m) = ln(\alpha_k(m)), \qquad \tilde{\beta}_k(m) = ln(\beta_k(m)), \quad and \qquad \tilde{\gamma}_i(\vec{\lambda}_k, m) = ln(\gamma_i(\vec{r}_k, m)) \tag{3.7}$$

The implications of the logarithm are that the multiplications change to additions (3.8 ...3.12), which are cheaper to calculate. Branch metrics now use LLRs for Soft Input ($\lambda_k = \frac{4 \cdot A \cdot r_k}{N_0}$).

$$\Lambda(d_k) = ln \frac{\sum_{m'=0}^{2^\mu - 1} e^{\tilde{\alpha}_k(m') + \tilde{\gamma}_1(\vec{\lambda}_k, m') + \tilde{\beta}_{k+1}(S_f^1(m'))}}{\sum_{m'=0}^{2^\mu - 1} e^{\tilde{\alpha}_k(m') + \tilde{\gamma}_0(\vec{\lambda}_k, m') + \tilde{\beta}_{k+1}(S_f^0(m'))}} \tag{3.8}$$

$where:$

$$\tilde{\alpha_k}(m) = ln \sum_{i=0,1} e^{\tilde{\alpha}_{k-1}(S_b^i(m)) + \tilde{\gamma}_i(\vec{\lambda}_{k-1}, S_b^i(m))} \tag{3.9}$$

$$\tilde{\beta_k}(m) = ln \sum_{i=0,1} e^{\tilde{\beta}_{k+1}(S_f^i(m)) + \tilde{\gamma}_i(\vec{\lambda}_k, m)} \tag{3.10}$$

$$\tilde{\gamma}_i(\vec{\lambda}_k, m) = i \cdot (\lambda a_k^s + \lambda i_k^s) + p(m,i) \cdot \lambda i_k^p \tag{3.11}$$

$$\lambda e_k = \Lambda(d_k) - (\lambda a_k^s + \lambda i_k^s) \tag{3.12}$$

where additionally:

$\lambda e(d_k)$	extrinsic soft output (Soft output without direct systematic information of bit k)
$\tilde{\alpha}_k(m)$	forward recursion state metric for state m in trellis step k
$\tilde{\beta}_k(m)$	backward recursion state metric for state m in trellis step k
$\tilde{\gamma}_i(\vec{\lambda}_k, m)$	branch metric if current state is m and input bit equals i, using LLR from trellis step k
$\vec{\lambda}_k$	Log-Likelihood Ratio's λi_k^s, λa_k^s and λi_k^p
λi_k^s	systematic intrinsic information of trellis step k
λa_k^s	systematic a-priori information of trellis step k
λi_k^p	parity intrinsic information of trellis step k
$p(m,i)$	parity bit if current state is m and systematic bit equals i

This algorithm is called the *log-MAP* algorithm. The BER performance of log-MAP is the same as for BCJR [19]. Extrinsic information (3.12) is calculated by subtracting the systematic LLR from the soft output Λ. This prevents systematic information produced by SISO 1 or SISO 2 being used as input for respectively SISO 1 or SISO 2 in the next iteration. If systematic information is reused, it will destabilise the feedback loop (which turbo decoding actually is). Systematic extrinsic soft information λe_k^s becomes systematic a-priori information λa_k^s after (de-)interleaving it. This a-priori information is added to the systematic intrinsic information (in the next SISO module), as can be seen in equations 3.12 and 3.12. The result of this addition is an indirect change of the decision function shown in Figure 2.5 on page 17, making BCJR type algorithms Maximum

A-posteriori Probability (MAP). Equation 3.13 is the only expensive operation remaining in the log-MAP algorithm.

$$ln \sum_{x} e^{p(x)} \tag{3.13}$$

This equation can be recognised in the Soft Output calculation, the forward recursion, and in the backward recursion. It is possible to simplify this equation using the Jacobian transformation. This transformation is shown in equation 3.14.

$$\begin{aligned} ln\left(e^{a}+e^{b}\right) &= \mathrm{Max}(a,b) + ln\left(1+e^{-|a-b|}\right) \\ &\approx \mathrm{Max}(a,b) + f_c(a-b) \doteq \mathrm{Max}\star(a,b) \end{aligned} \tag{3.14}$$

Correction function $f_c(x)$ can be implemented as a look-up table. For a quantised implementation this look-up table correction function can be an exact implementation form of $ln\left(1+e^{-|a-b|}\right)$. In general it is an approximation.

If equation 3.13 is substituted into equations 3.8, 3.9, and 3.10 the algorithm is called *Max⋆-log-MAP*. The Max⋆-log-MAP can be described using the following equations:

$$\Lambda(d_k) = \underset{m'=0}{\overset{2^{\mu}-1}{\mathrm{Max}\star}} \left(\tilde{\alpha}_k(m') + \tilde{\gamma}_1(\vec{\lambda}_k, m') + \tilde{\beta}_{k+1}(S_f^1(m'))\right) \tag{3.15}$$

$$- \underset{m'=0}{\overset{2^{\mu}-1}{\mathrm{Max}\star}} \left(\tilde{\alpha}_k(m') + \tilde{\gamma}_0(\vec{\lambda}_k, m') + \tilde{\beta}_{k+1}(S_f^0(m'))\right) \tag{3.16}$$

$$\tilde{\alpha}_k(m) = \underset{i=0,1}{\mathrm{Max}\star} \left(\tilde{\alpha}_{k-1}(S_b^i(m)) + \tilde{\gamma}_i(\vec{\lambda}_{k-1}, S_b^i(m))\right) \tag{3.17}$$

$$\tilde{\beta}_k(m) = \underset{i=0,1}{\mathrm{Max}\star} \left(\tilde{\beta}_{k+1}(S_f^i(m)) + \tilde{\gamma}_i(\vec{\lambda}_k, m)\right) \tag{3.18}$$

$$\tag{3.19}$$

The other algorithm from the BCJR family is the *Max-log-MAP* algorithm. The only difference between Max-log-MAP and Max⋆-log-MAP is the negligence of the correction factor (equation 3.14) in the first algorithm. *Max⋆-log-MAP* and *Max-log-MAP* are the two important implementation variants of the BCJR algorithm.

Section 3.2.1 explained that the BCJR algorithm has a forward and a backward calculation. These calculations are performed by the equations 3.17 and 3.18 respectively. Soft Output is calculated in equation 3.15. It subtracts the logarithm of transition values, leading to a zero as output bit, from the logarithm of transition values, leading to a one. The transition value is the sum of state metrics and the branch metric, belonging to a transition. This is more advanced than finding the highest sum of state metrics, belonging to a transition, suggested in section 3.2.1.

3.2.3 Complexity and performance

In this section the complexity and performance of the BCJR algorithms are discussed. A detailed cost analysis is shown in Section 4.4. For comparison with modified SOVA the figures of the implemented NII architecture, explained in section 4.4.4, are used.

cost i. Operations: $[3 \cdot 2^n - 3 \cdot n + 4 \cdot 2^\mu \cdot (1 - 2^{-n}) + 6 \cdot 2^\mu - 18]$ additions and $[4 \cdot 2^\mu - 2]$ 'compare selects' (maximum operation)

cost ii. State Metrics Memories: $\left[(2 \cdot \frac{B}{W} + W) \cdot (2^\mu - 1) \cdot (P_s + 1)\right]$

cost iii. Branch Metric Memories: $[W \cdot (P_{\lambda^s} + P_{\lambda^p})]$

cost iv. Registers:$[2 \cdot (2^\mu - 1) \cdot (P_s + 1) + P_{\lambda^s} + P_{\lambda i} + P_{\lambda a}]$

The figures for the 3GPP configuration are:

- μ = number of delay elements in the RSC, μ=3
- B = block-length, B=5120
- W = window-length, W=$\lceil\sqrt{B}\rceil$=72
- n = code rate $\frac{k}{n} = \frac{1}{2}$, i.e. n=2
- P_s = precision of state metric, 7 bits
- $P_{\lambda a}$ = precision of a-priori LLR, 6 bits
- $P_{\lambda i}$ = precision of intrinsic LLR, 4 bits
- $P_{\lambda^s} = \lceil^2 log(2^{P_{\lambda a}} + 2^{P_{\lambda i}})\rceil$ = precision of systematic LLR, 7 bits

The official maximal block length in 3GPP is set to 5114, but for simplicity the general block length 5120 is used. The presented figures result into the next implementation cost:

cost i. operations: 60 Additions, 30 'compare selects'

cost ii. state metric memories: 12096 bits

cost iii. branch metric memories: 792 bits

cost iv. registers: 129 bits

When using compressed NII implementation techniques, state metric memory can be reduced. If the compressed state metric vectors (μ bits) are saved in the same memory as the state metric vectors ($(2^\mu - 1) \cdot (P_s + 1)$ bits), stake memory can be reduced with a factor $R_f = \lfloor\frac{(2^\mu - 1) \cdot (P_s + 1)}{\mu}\rfloor = 18$. Note that the floor is taken under the assumption that each state metric vector has its own memory address, and compressed state metric vectors are not saved on more than one memory address. Under these conditions, the minimal memory requirements are $\lceil\sqrt{\frac{8 \cdot B}{R_f}}\rceil \cdot (2^\mu - 1) \cdot (P_s + 1) = 2688$ bits, and a window length equal to 24. For a window-length of minimal 40, state metric memories sum up to:

$$(\lceil\frac{2}{R_f} \cdot \frac{B}{40} + 40\rceil) \cdot (2^\mu - 1) \cdot (P_s + 1) = 3080 \text{ bits} \tag{3.20}$$

The implementation cost of compressed NII are 7 compare select operations and a vector reconstruction mechanism. For performance of the BCJR algorithms refer to appendix A.

Complexity evaluations for derived log-MAP algorithms have been performed at Philips Nat.lab. Crozier [4] shows that performance degradation of a scaled Max-Log-MAP is within 0.2 dB of Max⋆-log-MAP. Simulations confirm this performance and narrow the difference to 0.1 dB for UMTS configurations. Recent breakthrough's in sliding window schemes allow full window BCJR performance against square root order memory implementation cost. These schemes will be explained in section 4.4.

3.3 Comparison SOVA/BCJR

As can be seen in section 3.1.2 and 3.2.3 there are 2 dimensions for comparing the algorithms: Implementation cost and performance. The BCJR-type algorithms are more expensive than the SOVA-type algorithms. This is due to the fact that BCJR is a sort of 2-way Viterbi algorithm, while SOVA is a 1-way algorithm. The operation cost of Max-Log-MAP is 10% $\cdots$ 20% more expensive, while the memory cost are 50% $\cdots$ 400% more expensive, compared to MSOVA. BCJR-type algorithms in general have a higher or equal performance compared to the SOVA-type algorithms. This is due to the fact that BCJR is theoretical optimal, while MSOVA is sub optimal. the SOVA algorithm with the highest performance and the BCJR-algorithm with the lowest performance are equal. The performance of the BCJR variant can be improved by scaling the extrinsic information. The influence of scaling the SOVA type algorithms is unknown.

The analysis report of H. Krauß[15] shows that, for some block lengths and $\frac{Eb}{N_0}$ the difference between the BCJR (Max⋆-log-MAP) and modified SOVA (Max-log-MAP) can be compensated by increasing the number of iterations of modified SOVA (e.g. 4 iterations Max⋆-log-MAP $\approx$ 10 iterations modified SOVA). This implies that, if early stopping policy (section 4.5) is used, then the BER differences will be negligible. Max⋆-log-MAP however will need less iterations, resulting in lower latency and lower power consumption. For the other block lengths and $\frac{Eb}{N_0}$, Max⋆-log-MAP will achieve performance, which can not be achieved using SOVA.

There are two aspects in our approach which has not yet been analysed. The first aspect is that for SOVA we went from a simple algorithm to complex algorithms, requiring more hardware. For BCJR we reversed the process, starting with the expensive algorithm and in several steps going to a cheaper to implement algorithm with the drawback of loosing performance. This automatically implies that when requiring a higher performance of the turbo decoder, BCJR can be adjusted easily (reversing some steps made), while SOVA needs major changes. The second aspect is the higher flexibility of the BCJR algorithms, concerning variable block lengths. SOVA has a survivor depth of 40 in case of a fading channel. If the block length of the code varies between this 40 and 5120, the algorithm performs sub-optimal for the lower block lengths. The same accounts for the sliding window with training calculations BCJR variant, explained in Section 4.4.3. Other implementation variants of the BCJR family, for example NII and EI-BCJR, have better adaption techniques to the block-length. When combining the stakes and state metric memories, a dynamic window control exists, where for small block lengths, full block techniques can be applied, and for larger block stakes can be added. This dynamic control field is further explained in Section 5.2.1 on page 48.

When working under high hardware/area constraint, SOVA will be a very good alternative. When requiring high performance BCJR will be the best alternative. In most cases however, a design space has to be explored and, combined with early stopping, BCJR is the better algorithm family.

Within BCJR-type algorithms there are two interesting algorithms, Max⋆-log-MAP and Max-log-MAP. Performance differences between these two algorithms can be reduced when scaling extrinsic information λe_k^s. This scaling is further explained in section 4.2 and in appendix A.5.

Chapter 4

Implementation aspects of a turbo decoder

In this chapter we will look at the implementation details for turbo decoding using Max($\star$)-log-MAP. The choice for these algorithms was motivated in Section 3.3. The aim of this report is to explore the design -, and cost space of a turbo decoder. Before this can be done, first the implementation aspects have to be explained. This will be the aim of this section, starting with quantization. Implementation techniques for the logarithmic correction factor and scaling a-priori information will be explained in section 4.2. During the forward and backward calculations the state metric vector has to be normalised. This normalisation is explained in section 4.3. Section 4.4 explains implementation techniques for the BCJR-type algorithms. This chapter is concluded with the explanation of the early stopping policy.

4.1 Quantization

The process of limiting the representation precision of a value is called quantization. For quantization: word-lengths (number of bits per word), rounding schemes, overflow characteristics, and quantization techniques as uniform, non-uniform, and logarithmic have to be considered. In this chapter mainly word-lengths are considered. These word-lengths are important, because the data path and memory cost scale approximately linearly with the word length. For rounding, schemes called 'rounding to infinity' ($-\frac{1}{2} \rightarrow 0, \frac{1}{2} \rightarrow 1$) and saturation as overflow characteristic (on overflow the result keeps the maximum or minimum value) are used. Another question is where to put the point in the word. Especially operations like multiplication and Max$\star$ are sensitive for scaling and this will have implications for the quantization scheme after the operation. Jeong [12] gives an extensive survey on input quantization techniques (uniform, non-uniform and logarithmic), word-lengths and clipping levels for turbo decoder implementation. Input quantization is needed to transform the time discrete input signal to values on which the digital turbo decoder can calculate. The conclusions of Jeong are:

- Uniform quantization
- 6-bit channel input, 8-bit extrinsic information
- Clipping levels of 2 and 8 times the amplitude sent, for channel input and extrinsic information respectively. This implies that the number of bits behind the point for both words are equal.

An implication of these figures is that the saturation levels are a factor 2 larger than the received signals which, if no channel fading is used, result in a linear mapping on the interval $[-2 \cdot A \ldots 2 \cdot A]$. Simulations show that performance differences between 6 bit channel input and 4 bit channel input are very small, while the implementation cost differ approximately a factor $\frac{6}{4}$. Jeong [12] also showed that the gain of making the a-priori LLR more than a factor 4 larger than the intrinsic LLR, gives negligible performance difference. For the

Max-log-MAP algorithm the difference between the largest and smallest state metric value can be calculated using the eigen-vector of the trellis, explained in appendix B. For both forward and backward calculations this maximum delta is 10 times larger than the maximum absolute intrinsic LLR value, under the assumption that the maximum absolute a-priori LLR is 4 times the maximum absolute intrinsic LLR. Further studies have to point out if this assumption is reasonable. Numerical calculations of the eigen-vector show that the maximum delta state metric value is smaller for the Max$\star$ variant, compared to the Max variant. It is therefore save to reserve the same word-space for both variants. This result will be used in the normalisation of the state metric vector, explained in section 4.3. Summarising the results:

- received signals are linearly mapped on a 4-bits LLR word, including one sign bit
- a-priori LLR use 6-bit words, including 1 sign bit and have the same number of bits behind the point compared to the received LLR
- the maximum difference between state metric values is 10 times the maximal absolute intrinsic LLR (3 bits) and therefore can always be saved in a 7 bits word ($P_s = 7$)

For the scaling variants it is wise to apply overflow characteristics after multiplication with the scaling factor.

4.2 Logarithmic Correction and scaling of extrinsic information

As explained in section 3.2.2 the Max$\star$-log-MAP algorithm has a Max$\star$ operation, which consists of a maximum operation and a correction factor $f_c(x)$ (equation 3.14). Floating point values for this function are shown in Figure 4.1. Since a fixed point with maximal 8 positive values (3 bits) is used, this correction factor strongly reduces in complexity. Figure 4.1 shows the correction function for floating point, 2, 3 and 5 level quantization.

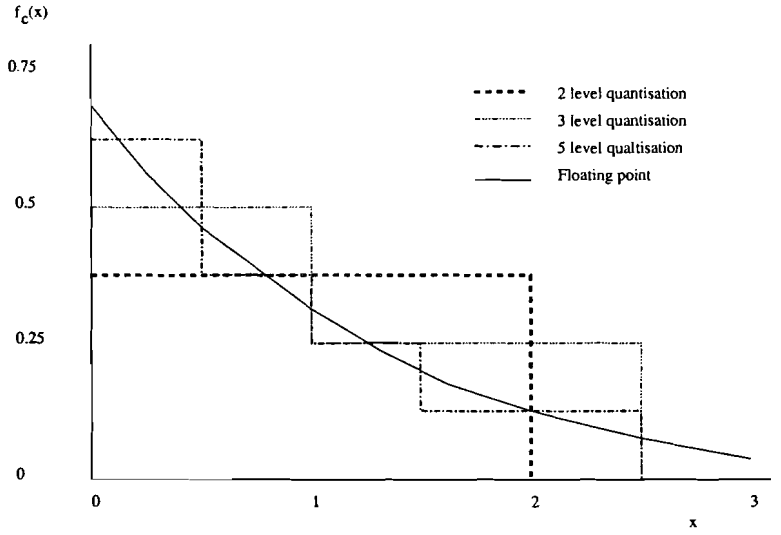

Figure 4.1: The logarithmic correction function

An 2 level quantization is proposed by Gross [8], who claims a loss of less than 0.03 dB compared with floating point simulations. Park [11] claims that a Max$\star$ with three levels of correction values can achieve about 0.1 dB lower than the performance of the optimal BCJR decoder, in spite of errors due to quantization. No attempt is made to explain the difference in BER performance between Gross and Park. The results are used as an indication that no large number of entries is required. Park and Gross use a 4/6/8 scheme for

quantization, which stands for 4 bit intrinsic LLR, 6 bit a-priori LLR and 8 bit state metric values.

A low number of correction factors can be implemented cheaply, using a lookup table or direct logic. The 2 level quantization, proposed by Gross, can be implemented using the architecture shown in Figure 4.2.

$$f(x) = \begin{cases} \frac{3}{8} & -2 \leq x < 2 \\ 0 & otherwise \end{cases}$$

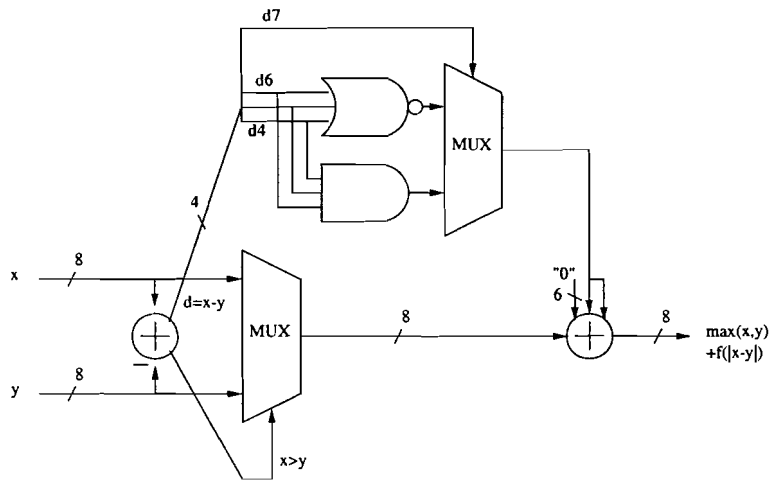

Figure 4.2: One bit logarithmic correction for 3 bits behind the point

Note that the inputs and outputs are 8-bit, a wrapped around overflow characteristic is used, 3 bits behind the point are assumed, and that an a-symmetric correction factor is used.

For 2 and 1 bit behind the point, the next correction tables implement the correction function optimally:

$$f_c(x) = \begin{cases} \frac{3}{4} & x = 0 \\ \frac{1}{2} & 0 < |x| < 1 \\ \frac{1}{4} & 1 \leq |x| < 2 \\ 0 & |x| \geq 2 \end{cases} \quad \text{or} \quad f_c(x) = \begin{cases} \frac{1}{2} & 0 \leq |x| \leq 1 \\ 0 & |x| > 1 \end{cases}$$

Crozier [4] proposed a different approach. He uses the Max-log-MAP algorithm, but multiplies the extrinsic information $Le(d_k)$ with a scaling factor $s_f = \frac{5}{8} = 0.625$. This multiplication can be implemented using basic operations like additions and shifts. Crozier claims a performance reduction of less than 0.2 dB compared to log-MAP. Simulations conducted for 3GPP configurations confirm a performance reduction of less than 0.15 dB. Appendix A.5 shows the simulation results. For some Eb/No the performance reduction is negligible. The following questions remain open:

- What are the best criteria to vary scaling factor s_f?
  (e.g. BER, WER, non linear, or number of iterations)
- Which scaling factors s_f achieve the best BER performance?
- What are the effects of scaling Max⋆-log-MAP?

One advantage of the Max-log-MAP approach is that the received signal does not have to be scaled with a factor $\frac{4 \cdot A}{N_0}$. Although the matching of this factor is not sensitive (a mismatch between -3dB and 6dB is allowed without significant degradation [12]) the performance will come closer to Max⋆-log-MAP. A scaled Max-log-MAP algorithm will be smaller and easier to implement compared to the Max⋆-log-MAP algorithm, due to the many Max⋆ operations. The Max⋆ operation is at least twice as expensive compared to the Max operation.

4.3 Normalisation

The state metric values, calculated by the forward and backward recursion can become very large, resulting in huge requirements for the number of bits in a state metric word. Normalisation is the technique for reducing the number of bits for this state metric word. State metrics are packed into a vector for each trellis step. This vector contains 2^{μ} state metric values in the following manner:

$$\alpha(k) = \begin{pmatrix} \alpha_0(k) \\ \alpha_1(k) \\ \vdots \\ \alpha_{2^{\mu}-2}(k) \\ \alpha_{2^{\mu}-1}(k) \end{pmatrix}$$

The maximal distance between 2 state metrics for the Max-log-MAP algorithm can be calculated using the eigen-vector technique, explained in appendix B. Numeric calculations show that the maximum distance for the Max$\star$ variant is smaller than the maximum distance for the Max variant. Due to the logarithmic character of the algorithms, it is possible to add a constant value to each state metric within a vector. This addition will not change the outcome of the algorithm. With this addition and the known maximum distance, it is possible to prevent the state metrics to grow and to use too many bits. This technique is called normalisation and there are 3 variants:

- using a wrapped around overflow characteristic
- subtracting the largest (or smallest) state metric from the others
- subtracting one previously determined state metric from the others

The first normalisation technique is used by Masera [16]. It uses the fact that, if a wrapped around overflow characteristic is used, then all values will lie on a circle. Figure 4.3 shows this circle for a 3 bit word.

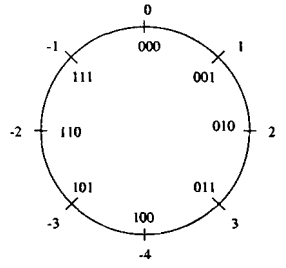

Figure 4.3: Example of metric representation

If the maximal difference between two values is less than half a circle it is alway possible to determine which value is the largest. For example:

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

Note that if calculating wrapped around, and the maximum difference is less than 4 (half the circle) then -4 is larger than 2.

The implications of wrapped around calculations with a maximum distance of half a circle are:

- no calculations are needed for normalising the forward and backward calculation
- each state metric word requires 1 bit more, compared to the word length calculated in section 4.1
- additional calculations are needed for the sign extension, required in the soft output calculation (direct addition of forward and backward state metrics will exceed the maximum difference of half the circle, loosing the property of knowing which one is the largest)

Since the forward and backward calculations are in a recursive critical path, it is very interesting that no calculations are needed for normalisation. The meaning of a recursive critical path is that both the calculations have to be conducted before the next trellis step can be calculated, and that they cannot be calculated fully in parallel with other calculations. The additional calculations, required in the soft output calculation are not important, because they are not in a recursive critical path. For this normalisation scheme and the quantization explained before, the next area cost are required:

I. state metric vector: $2^{\mu} \cdot (P_s + 1)$ (=64) bits

II. operations : $2 \cdot (2^{\mu} - 1)$ (= 14) subtractions if sign extention has to be applied in the Soft Output unit

Note that the $2 \cdot (2^{\mu} - 1)$ subtractions are not required if the sign extention is not needed.
The second normalisation technique consists of subtracting the largest (or smallest) state metric value from the others. This is a very simple straight forward implementation technique, with quite high logic area cost and lower memory requirements:

I. state metric vector: $2^{\mu} \cdot P_s$ (= 56) bits

II. operations : $2 \cdot (2^{\mu} - 1)$ (= 14) maximum operations for finding the maximum state in the soft output calculations and $2 \cdot (2^{\mu} - 1)$ (=14) subtractions.

The main disadvantage of this scheme is the large number of operations needed in the recursive critical path. A solution, combining the advantage of the previous two solutions, is subtracting a previously determined state metric, for example $\alpha_0(k)$ form each forward recursion state metric, and $\beta_0(k)$ from each backward recursion state metric. The implementation cost of this solution, invented by J. Dielissen, are:

I. state metric vector: $(2^{\mu} - 1) \cdot (P_s + 1)$ (= 56) bits

II. operations : $2 \cdot (2^{\mu} - 1)$ (= 14) subtractions.

Note that for memory requirements of this solution and the previous one, are equal if $2^{\mu} \cdot P_s = (2^{\mu} - 1) \cdot (P_s + 1)$ which is the case. The number of additions can be reduced with 8, because of the property that $\alpha_0(k)$ and $\beta_0(k)$ are always zero. This solution is shown in Figure D.1. A disadvantage of this solution is that, for Max-log-MAP, the subtraction is after the Max operation in the recursive critical path. There are two feasible solutions to solve this disadvantage. The first solution is scaling the input vector instead of the output vector, resulting in a 2^{μ} (= 8) bit larger state metric vector. The second solution is propagation of the last subtractions into the previous maximum operation. This propagation is shown in figure D.2. The total implementation cost of this last normalisation technique are:

I. state metric vector: $(2^{\mu} - 1) \cdot (P_s + 1)$ (=56) bits

II. operations : $2 \cdot 2 \cdot (2^{\mu} - 1)$ (= 28) maximum operations

Note that for this last solution the property that $\alpha_0(k)$ and $\beta_0(k)$ are always zero, results in 8 additions in the data-path which can disappear. This solution is approximately 6 operations more expensive than the first proposed alternative. Since saving 8 bit, and doing 1 operation are approximately equal expensive the last proposed solution is the cheapest one if more than 6 state metric vectors have to be saved. For 3GPP this number varies between 40 and 216. Note that, for delay-times smaller than the clock cycle time, the solution with 56 bit, and 14 subtractions (and saving 8 additions) is always the cheapest. This normalisation technique is used in the design explained in section 5.2.2.

4.4 BCJR-type implementation techniques for SISO decoder

There are several implementation techniques for the BCJR-type algorithms [6]. This section explains the four main techniques. The first two techniques are true BCJR-type algorithms, the last two represent approximations of these algorithms. For each implementation technique, numerous calculation orders are possible. For the basic calculation order we can start at the beginning of the trellis, at the end of the trellis, or simultaneously at the beginning and end of the trellis.

For explaining the implementation cost the following 3GPP specific figures are used:

- μ = number of delay elements in the RSC, μ=3
- B = block-length, B=5120
- W = window-length, W=$\lceil\sqrt{B}\rceil$=72
- n = code rate $\frac{k}{n} = \frac{1}{2}$, i.e. n=2
- P_s = precision of state metric, 7 bits
- $P_{\lambda a}$ = precision of a-priori LLR, 6 bits
- $P_{\lambda i}$ = precision of intrinsic LLR, 4 bits
- $P_{\lambda s} = \lceil^2 log(2^{P_{\lambda a}} + 2^{P_{\lambda i}})\rceil$ = precision of systematic LLR, 7 bits

4.4.1 Full block technique

First a full block implementation technique is explained, whose schedule is visualised in Figure 4.4. Forward and backward recursion are carried out over the full block of data. The backward recursion starts after the last forward recursion. The Soft Output calculation starts immediately after the calculation of the corresponding backward state metric vector in the backward recursion. The corresponding forward state metric vector is retrieved from the memory in which it was saved during forward recursion.

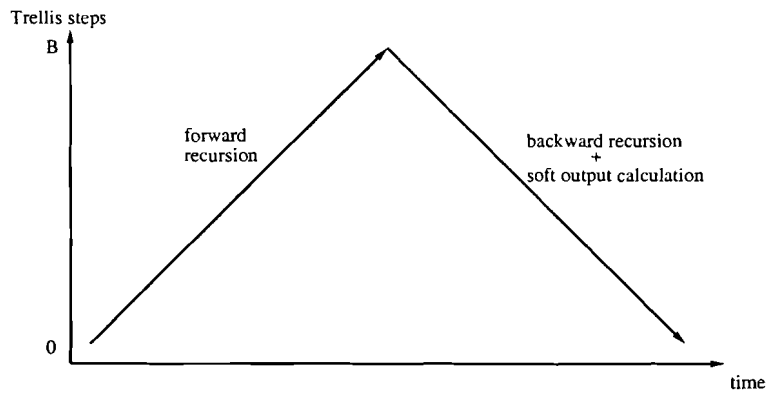

Figure 4.4: full block BCJR

This implementation technique is mathematically equivalent to the BCJR-type algorithm. The results for implementation cost of this technique are:

cost i. Operations: $[3 \cdot 2^n - 3 \cdot n + 4 \cdot 2^\mu \cdot (1 - 2^{-n}) + 6 \cdot 2^\mu - 18]$ additions and $[4 \cdot 2^\mu - 2]$ compare selects
60 additions, 30 compare selects (per output bit)

cost ii. State Metrics Memories: $[B \cdot (2^\mu - 1) \cdot (P_s + 1)]$
286720 bits

cost iii. Branch Metric Memories: dual retrieval from background memory

Dual retrieval means that the branch metric values are read twice from the memory. This dual retrieval does not cost any chip area, but will cost more power dissipation.
Note that a very large number of state metrics bits have to be saved. If a trellis step can be calculated in one clock cycle the latency of this algorithm is of order $2 \cdot B$ $(= 10240)$ clock cycles.

4.4.2 Efficient Implementation BCJR

The schedule of the second true BCJR-type algorithm, Efficient Implementation-BCJR (Figure 4.5) starts with the backward recursion, and saves stakes. Stakes are metric vectors at trellis-steps B, B-W, ..., W. When the forward calculation reaches a stake, the backward recursion is initialised with the corresponding backward recursion stake. Soft output calculations start immediately after the calculation of the corresponding backward recursion state metric vector. This true BCJR-type algorithm is named for its efficient state metric vector memory, which is reduced to the minimum of order square root of the block-length relative to the linear memory requirements of full block BCJR.

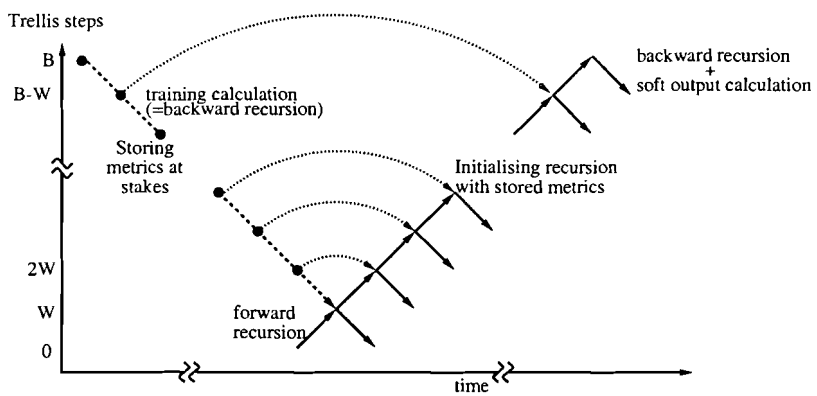

Figure 4.5: Efficient Implementation BCJR

This minimum is reached if $W = \sqrt{B}$ and can be calculated using the equation for calculating state metric memories, stated below.

cost i. Operations: $[3 \cdot 2^n - 3 \cdot n + 6 \cdot 2^\mu \cdot (1 - 2^{-n}) + 7 \cdot 2^\mu - 19]$ additions and $[5 \cdot 2^\mu - 2]$ compare selects
79 additions, 38 compare selects (per output bit)

cost ii. State Metrics Memories: $[(\frac{B}{W} + W) \cdot (2^\mu - 1) \cdot (P_s + 1)]$
8064 bits

cost iii. Branch Metric Memories: $[W \cdot (P_{\lambda^s} + P_{\lambda^p})]$
792 bits and dual retrieval

The memory for saving state metrics bits is reduced to $2 \cdot \sqrt{B} \cdot (2^{\mu} - 1) \cdot (P_s + 1)$, which is of order square root. This memory optimisation has an implementation cost of 19 additional additions and 8 additional compare selects. If a trellis step can be calculated in one clock cycle, then the latency of this algorithm is of order $2 \cdot B - W$ (= 10168) clock cycles.

4.4.3 Sliding window technique with training calculations

Figure 4.6 shows the schedule of sliding window with training calculations. This is the first non-true BCJR-type algorithm. In this technique, the backward recursion is not initialised with the exact state metric vector, but with an approximation. Several steps ahead in the trellis, the training recursion is started with an uniform state metric vector. After several training calculations the state metric vector converges to an estimation of the correct state metric vector. When conducting $5 \cdot (\mu + 1)$ training calculations, the performance loss is negligible for an AWGN channel. μ is the number of delay elements in the RSC (for 3GPP $\mu = 3$). For Rayleigh fading channel characteristics $10 \cdot (\mu + 1)$ training calculations need to be calculated. Note that training calculations imply both more silicon area and more power dissipation.

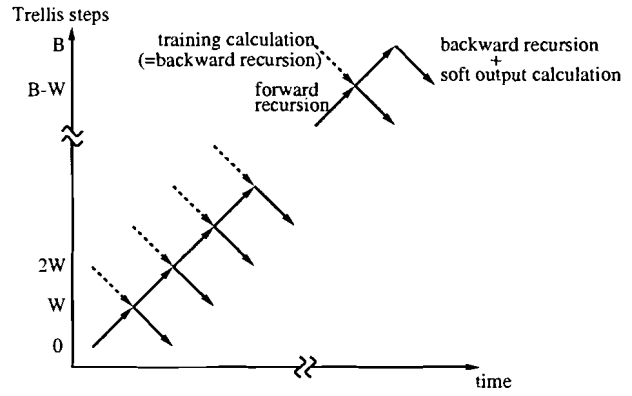

Figure 4.6: Sliding window with training calculations

cost i. Operations: $[3 \cdot 2^n - 3 \cdot n + 6 \cdot 2^{\mu} \cdot (1 - 2^{-n}) + 7 \cdot 2^{\mu} - 19]$ additions and $[5 \cdot 2^{\mu} - 2]$ compare selects
79 additions, 38 compare selects (per output bit)

cost ii. State Metrics Memories: $[W \cdot (2^{\mu} - 1) \cdot (P_S + 1)]$
4032 bits

cost iii. Branch Metric Memories: $[2 \cdot W \cdot (P_{\lambda^s} + P_{\lambda^p})]$
1584 bits

State metric memory is reduced with a factor 2, while the branch metric memory is doubled. Note that no double retrieval from the background memory is required, resulting in half the input memory bandwidth. The number of calculations is equal to the EI-BCJR algorithm. The latency of this algorithm is of order $B + W$ (= 5192), a reduction of 50 %, compared to the two previously shown alternatives.

4.4.4 Sliding window Next Iteration Initialisation

The Next Iteration Initialisation (NII) implementation technique [6] is shown in Figure 4.7. The backward recursions at the stakes are initialised with stakes from the previous iteration. Note that SISO 1 can only be initialised with stakes from SISO 1, resulting in twice the amount of stake memory, compared to EI-BCJR.

In the first iteration of this technique, either EI-BCJR initialisation can be used, training calculation can be performed, or an uniform state metric vector can be used for initialisation.

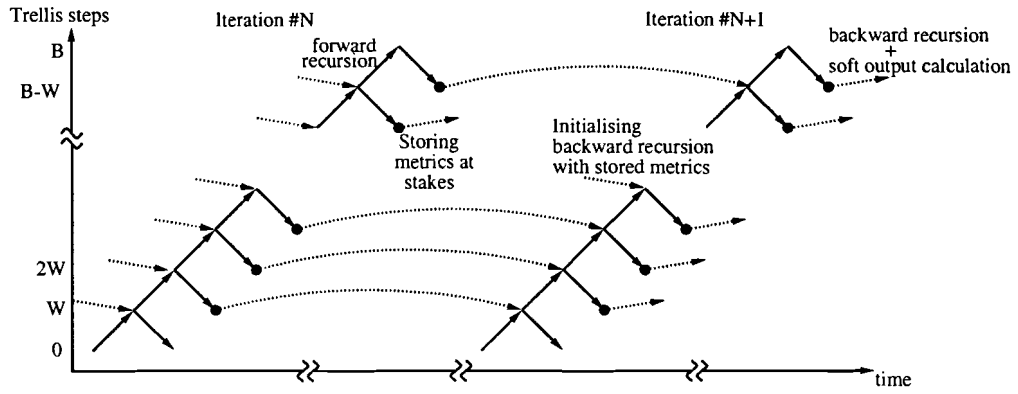

Figure 4.7: Sliding window Next Iteration Initialisation

cost i. Operations: $[3 \cdot 2^n - 3 \cdot n + 4 \cdot 2^\mu \cdot (1 - 2^{-n}) + 6 \cdot 2^\mu - 18]$ additions and $[4 \cdot 2^\mu - 2]$ compare selects (if an uniform state metric vector is used for initialisation)
60 additions, 30 compare selects (per output bit)

cost ii. State Metrics Memories: $\left[(2 \cdot \frac{B}{W} + W) \cdot (2^\mu - 1) \cdot (P_s + 1)\right]$
12096 bits

cost iii. Branch Metric Memories: $[W \cdot (P_{\lambda^s} + P_{\lambda^p})]$
792 bits

This algorithm requires the same number of operations as the full block algorithm. State metric memory however is an order square root of block length smaller. This memory requirement is three times the state metric memory requirement for sliding window with training calculations, and $1\frac{1}{2}$ times the state metric memory requirement for EI-BCJR. This is due to the stakes, which have to be stored for SISO 1 and SISO 2. The requirements for branch metric memories are half of the requirements for sliding window with training calculations. The latency of this algorithm is of order $B + W = 5192$, equal to the previous variant. Note that for this algorithm no double branch metric memory bandwidth is required for the first window. The bandwidth to the state metric memory increases with $\frac{1}{W} = \frac{1}{72}$ due to saving stakes.

A slight modification to NII is reverse order NII, a schedule where all odd iterations of the turbo decoder are calculated with the forward recursion calculations as the main stream and all even iterations with the backward recursion calculations as the main stream. The advantage of such a scheme is that no calculations from more than 1 iteration ago are used.

Because of the large number of stakes which have to be saved and the number of bits required for each stake, the cost of NII become high. For block length 5120, window length 72, state metric vector of 56 bit, the memory cost sum up to a total of 12096 bits which have to be saved. Vector reduction is one way to reduce the number of bit. The lowest number of bits which have to be saved is μ, representing the number of the state having the highest probability. The implementation cost of such a reduction scheme are $2^\mu - 1$ maximum operations. The options for vector reconstruction and the influence of this vector reconstruction on the performance of a turbo decoder are explained in appendix A.4.

4.5 Early stopping

In most cases a hard stop criterion (e.g. 10 iterations) is built in to stop the iterations in the turbo decoder. The latency and power cost of conducting more iterations are considered too large, compared to the gain which can be expected. Early stopping is a policy that will stop the iteration process of the turbo decoder if the output is expected to be correct before the hard stop criteria is met. This early stopping policy will reduce both the average latency, and the power cost of decoding a block of data. Several options exist for early stopping:

1. The encoder supplies an error detection mechanism (e.q. checksum) and the turbo decoder detects if the message is correct with high probability.
2. The decoder calculates a checksum over the message and if the checksum is equal over 2 iterations, the message is equal to the previous iteration and might be correct.
3. Comparing the decisions of the previous SISO unit with the current output.
4. The decoder uses heuristic techniques to find out if the message has been converged. The message might be correct.

The problem with the first option is the required alteration of the encoder (not generally applicable). Other disadvantages are the lower rate of the code and the fact that some error detection schemes use the order of the sequence, making then not applicable in the second SISO unit.

The drawback of the second approach is that the same error in two following iteration remains undetected.

The drawback of the third approach is that the decisions of a full block have to be saved and the possibility that SISO 1 and SISO 2 make the same mistake for the same bits. For larger block lengths this possibility is very unlikely. Note that the decision might have to be saved for output production of the turbo decoder anyway.

In the heuristic approach, criteria are suggested to estimate if the message is stable. Wang et. al [24] suggests 4 easy to implement criteria, claiming an iteration reduction of more than 60% at Eb/No = 3.0, 8 iterations hard stop and block length of 1300 and 4300 bits. Their criteria are :

1. The absolute value of each extrinsic information $\lambda e(d_k)$ bit (3.12) is compared with a pre-set bound. Check if all of them are larger than the bound
2. The absolute value of each Soft Output bit $\Lambda(d_k)$ (3.15) is compared with a pre-set bound. Check if all of them are larger than the bound
3. The sign of each $Le(d_k)$ is compared with that of $\Lambda(d_k)$ at the same time instance k. Check if all of them within the current decoding block are identical
4. The number of 1's from the output of the current decoding block are accumulated. Check if the count from the previous iteration is the same as that from the current iteration.

The BER performance between hard stop and dynamic stopping, is the probability that convergence has been detected while de message is not correct and further iterations corrected these errors. Relaxing condition 3 (e.g. less than 3 differences are allowed) is a form of enlarging the error probability, resulting in less iterations.

Chapter 5

Turbo decoder design and implementation exercise

This chapter explains the design and implementation of key elements of a turbo decoder. The result of the design and implementation is a view of how a turbo decoder can be implemented, and what the implementation cost are. Not only will figures of area, delay, and power be shown, but also an explanation of how they are calculated and the architecture which is used. First the design methodology used is explained. Then implementation constraints and decisions are stated. Section 5.2 explains how a flexible architecture can be implemented. In section 5.3 the verification of the designed turbo decoder is discussed. This chapter concludes with the implementation cost of the turbo decoder.

5.1 Design methodology

Because of complexity the design-flow is divided into a number of steps [22]. In each step the basic principle of an application stays the same. Only the level of abstraction reduces in each step. An overview of the different levels of abstraction is shown in figure 5.1.

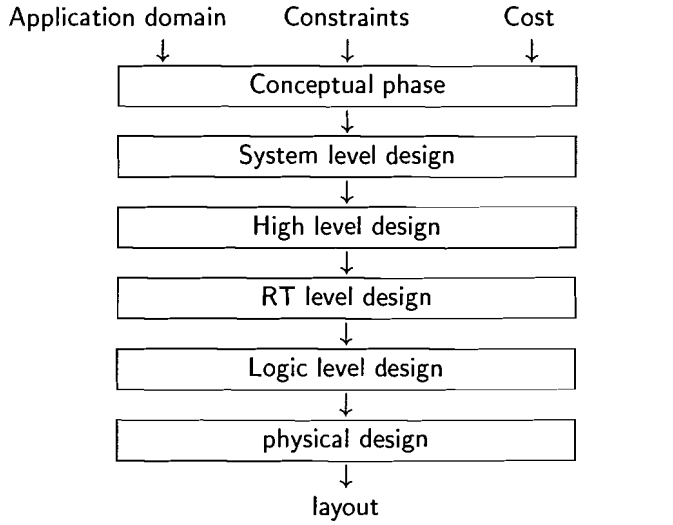

Figure 5.1: Top-down design flow

In this section we explain how the turbo decoder is designed, using the levels of abstraction. The levels of abstraction are divided into three groups. Within each group the same tools can be used for the different abstraction levels it contains.

5.1.1 Conceptual phase and System level design

During the conceptual phase the designer tries to reduce uncertainty. The results of this phase are a specification and a concept architecture. During this phase, turbo decoding for 3GPP and algorithms for SISO decoding are considered. Constraints from the 3GPP standard are taken into account and the decision for BCJR type SISO decoding is taken. A C++ specification is written, using parallel processes. For this C++ specification the Y-chart Application Programming Interface (YAPI) [5] is used. YAPI environment is based on Kahn modelling techniques: processes connected with each other using first in first out (FIFO) communication channels. YAPI is an additional library for the C++ language and can be combined with the A|RT library tool from Frontier Design [1] to achieve fixed point models.

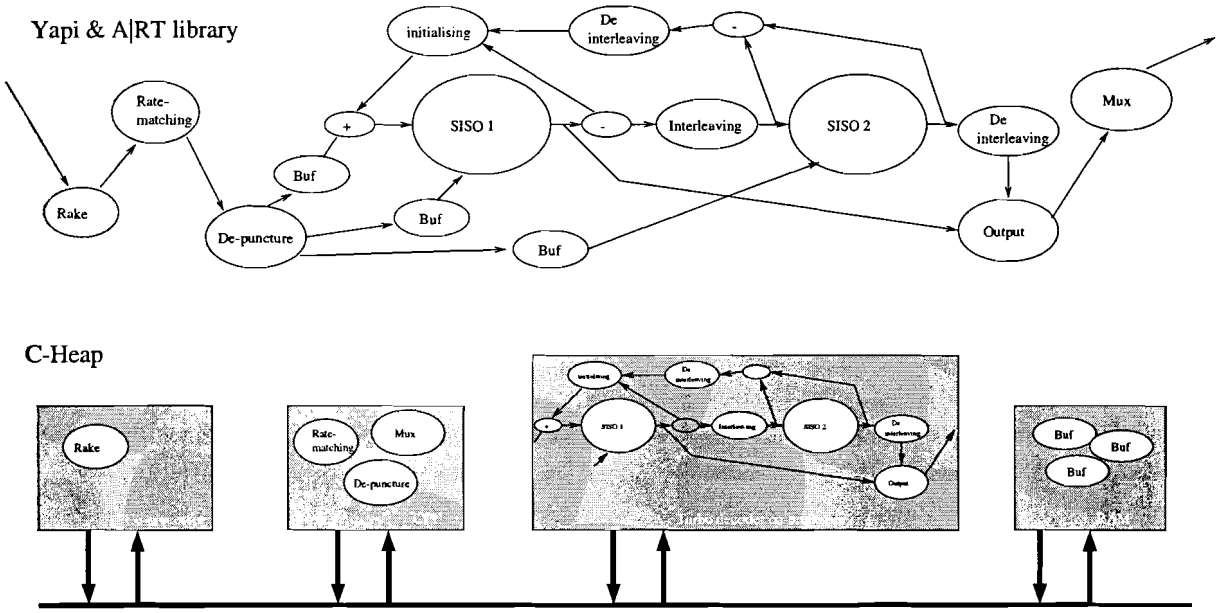

Figure 5.2: YAPI/A|RT library to C-Heap

Figure 5.2 shows the YAPI/A|RT library environment. This concept specification of the design is mapped to a C-HEAP [17] co-processor architecture. The advantages of implementing turbo decoding as a stand alone co-processor, which runs independently are the reduced complexity for the environment and possibilities to see the turbo decoder as an IP-block, enabling reuse. A co-processor can be seen as a black box with the interfaces shown in Figure 5.3.

When looking to the turbo decoder as a co-processor, the next question is where to put the interface between the turbo decoder and the environment. It is chosen to put the interface of the turbo decoder directly after the input memory and before the output memory.

[1] http://www.frontierd.com

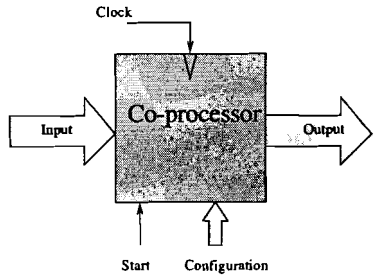

Figure 5.3: Co-processor interface

The reason for this decision are:

1. Reuse of the 'large' memories in other blocks
2. For different environments, different memory schemes are possible
3. Multi turbo decoder architectural implications

These multi turbo decoder architectural implications are explained in two steps. First of all a specific implementation is described, then the more general turbo decoder architecture is shown. Imagine a turbo decoder, where approximately every clock cycle a new input is presented (3·4 bits is assumed), and each block required P $\frac{1}{2}$ iterations. A turbo decoder can conduct a $\frac{1}{2}$ iteration in approximately block length clock cycles. The solution, presented in Figure 5.4, will give a very efficient implementation technique. Only one address generation unit is required, one global control is required, and wide word techiques can be applied.

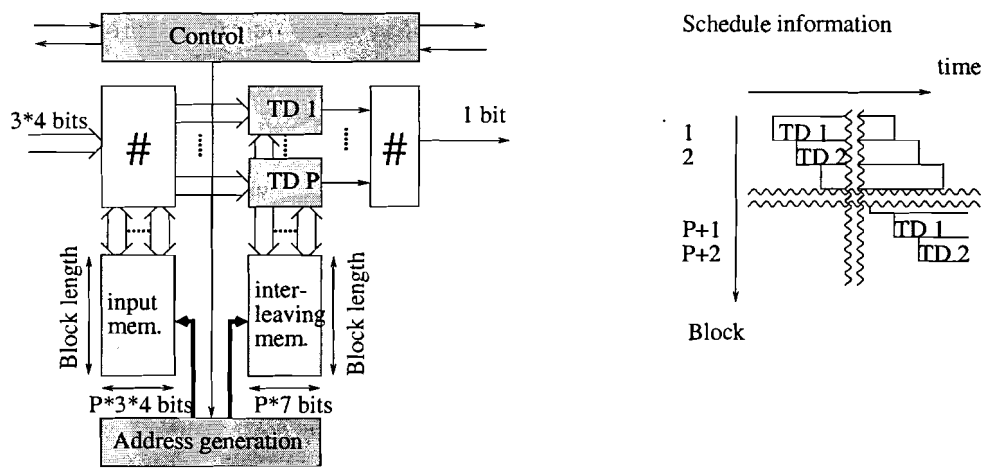

Figure 5.4: Specific multi turbo decoder implementation

The general (multi) turbo decoder architecture is shown in figure 5.5. This multi turbo decoding architecture, which enables very high speed turbo decoding, is the result of internal discussion at Philips Nat.Lab. A patent disclosure is written for this architecture.

The different turbo decoder co-processors run iterations at the same timing interval. The first advantage of this same interval processing is the use of a single (de-)interleaving address unit and wide word memory techniques. For input memory and (de-)interleaving memory, wide word techniques can be used to enlarge the

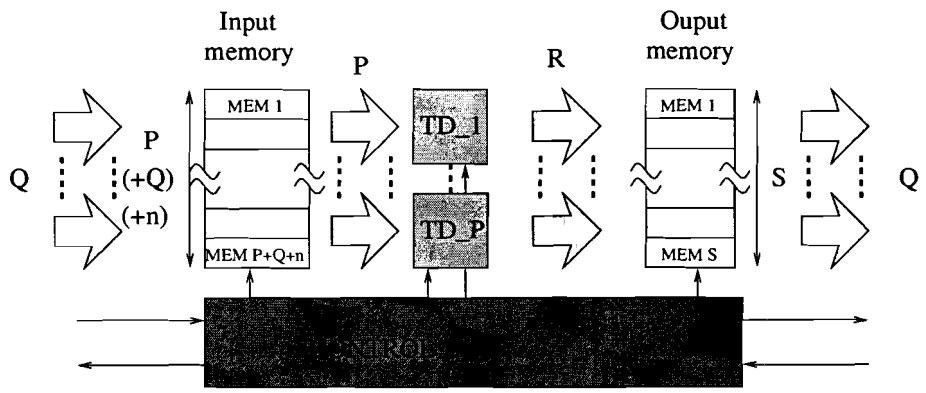

Figure 5.5: Multi turbo decoder implementation, using early stopping policy

re-usability of this memory, and to minimise the address overhead. The memories required for (de-)interleaving are inside the turbo decoder, while the input memories are external. The second advantage of this parallel turbo decoding scheme is the better utilisation of average processing time for each block. An algorithm, using early stopping techniques, can utilise this property better than serial concatenation of turbo decoders.

Q data blocks arrive each SISO decoding interval, where a SISO decoding interval is the time required by one SISO unit to decode one block of data. P turbo decoders are processing on P input memory blocks. If the data does not arrive during the same timing as reading the data for the SISO module, Q additional input memories might be required. If the difference in number of SISO decoding intervals is large, and processing is not possible, n additional input memory can be added (queueing). Depending on the cost relation of input memory and the turbo decoder, and the distribution of calculation time, P and n can be chosen. If, during operation, blocks arrives, while the P+n memories are full, then an algorithm appoints turbo decoders to stop iterating and continue with a new block. This algorithm uses early stopping criteria. Q has to be smaller or equal to P divided by the average number of processing blocks ($\frac{1}{2}$ iterations).

Note than at most P streams enter the turbo decoder array, while R streams leave the turbo decoder array. R is at most equal to P, and on average equal to Q. If there is no processing requirement for P turbo decoders, some of the turbo decoders can be turned off to save power. The output of the turbo decoders is buffered in an output memory. This output memory is smaller than the input memory due to the fact that only 1 hard decision bit instead of 3 soft input bits have to be saved. The aim of this output memory is to ensure that the output of the (multi) turbo decoder architecture has the same order as the input stream and to smoothen the output stream. In some cases no output memories are required and they can be removed from the architecture. In section 5.1.2 this turbo decoder co-processor is further explored.

5.1.2 High level design and RT level design

The next steps in the design flow concern the high level design. During the high level design of the turbo decoder co-processor, 3 layers of processing are recognised:

1. a flexible layer
2. a data flow layer for high throughput
3. an execution layer for executing the data path operations

These 3 layers of processing can be recognised in Figure 5.6:

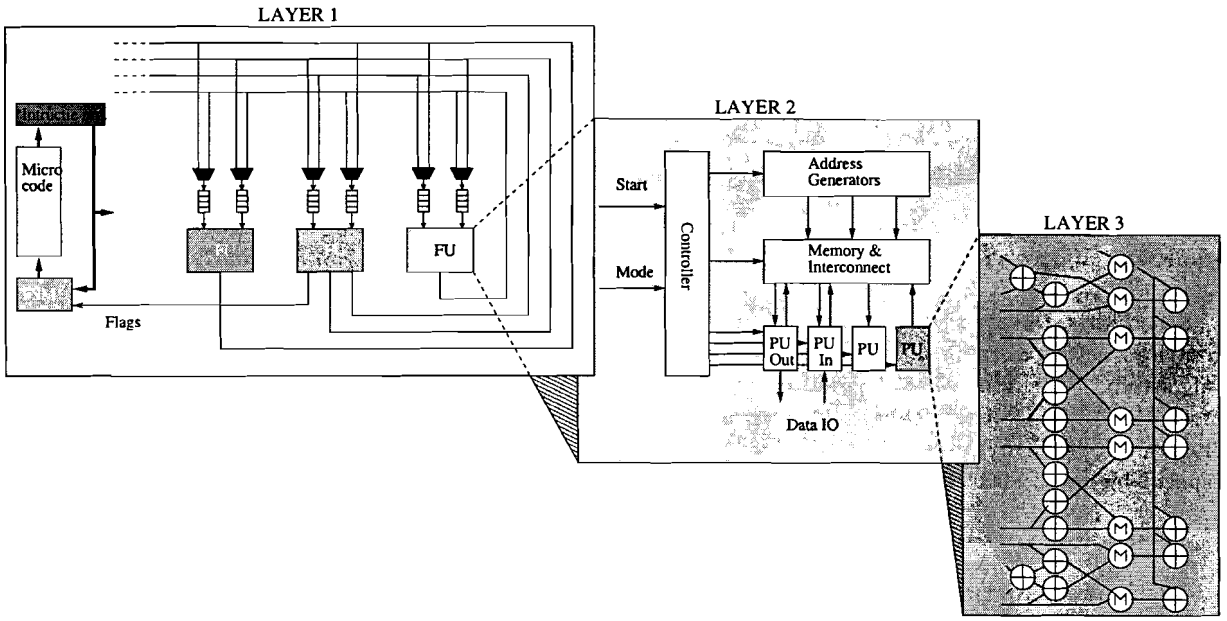

Figure 5.6: 3 layers of processing in a turbo decoder

In the **flexible layer**, control flow issues have to be handled. The following exceptions/irregularities are solved in this layer:

- Issues related to standardisation, e.g. 3GPP
  - trellis termination
  - variable block length adaption
  - SISO 1/SISO 2 swapping
  - interleaving/De-interleaving swapping
- Options in implementation, which influence performance and cost
  - vector compression/reconstruction
  - combined NII/cNII schedules
  - multi SISO module issues
  - manipulating extrinsic information using additional a-priori information
- Quality of service
  - programmable stopping criteria
  - early stopping
  - window length control

Depending on the number of items which play a role, different strategies can be applied. First of all this layer can be implemented by writing a shell manually in, for example, VHDL. This is a time consuming task, hampering the design of derivatives in the future. A second strategy is using a data flow architecture like PHIDEO [23]. The PHIDEO architecture is further explained in the data flow layer. This strategy is more flexible for future design and is not very time consuming. Due to the data flow architecture, it will however not provide an optimal solution. The third strategy is to use an application specific instruction set processor,

for example created using A|RT Designer from Frontier Design. A|RT Designer has a very long instruction word architecture where all functional units can be programmed in parallel. The architecture template of A|RT Designer is shown in the highest layer of processing in Figure 5.6.

A|RT Designer has a very large instruction word (VLIW) architecture. The program code is stored in a micro code memory. After defining a initial architecture, flexibility can be programmed. If needed, more units can be added to the architecture. This architecture can be future proof, will have a short implementation time, and will provide an efficient solution for the flexible control and various implementations.

The A|RT Designer architecture template is not optimal for high throughput processing like the processing required in the **data flow layer**. This second layer essentially consists of the sliding window BCJR-type algorithm: a streaming based signal processing algorithm. This layer is suited for a high throughput DSP tool like PHIDEO. The architecture template used in PHIDEO is shown in the second layer of processing in Figure 5.6. PHIDEO uses a PHIDEO Input Format (pif) as input format. This pif format is a language for specifying operations, their data dependencies, and additional constraints. Additional constraints can be:

- schedule constraints
- operation to data-path assignment
- signal to memory assignment
- memory addressing

An example of this pif file is shown in Figure C.2. This input file describes the algorithm and can be compiled to RT Level VHDL. A testbench can be generated automatically.

The remaining complexity for the lowest layer, the **execution layer**, is quite low. When operations have to be mapped onto a lower number of execution units, additional tools like FACTS [21] can be used. For the turbo decoder 1 execution unit for each operation is reserved. This is done for speed issues because each operation has a data introduction interval of 1 clock cycle. For the soft output calculation unit pipelining is optional. The forward and backward calculations have to be that fast because they are in a recursive loop: the next calculation can start only after finishing the current calculation. For implementing the execution unit there are 2 options: using A|RT library to convert C++ to VHDL/Verilog, or writing VHDL/Verilog manually.

After applying all the tools, a register transfer level architecture and a boolean level behaviour are available. The correctness of the output can be verified using the simulation. The fifo's of the YAPI model can be monitored. For the data flow layer verification the input values of the SISO module are used as input for a simulation tool, the output of the simulation tool can be compared with the output values of the SISO module. PHIDEO provides the full test-bench facilities for these tests. Simular verification techniques can be applied on the flexible turbo layer. Section 5.1.3 will use the verified boolean behaviour as an input.

5.1.3 Logic level and physical design

The boolean level VHDL description from the register transfer level design can be used for 2 different implementation techniques:

- Field Programmable Gate Arrays (FPGA)
- Application Specific Integrated Circuit (ASIC)

There are several large manufacturers for FPGA's for example Xilinx and Altera. FPGA's have a low time to market, but have at this moment a performance gap of a factor 3 à 5 and an area gap of a factor 10 à 30 compared to ASIC design. Due to memory requirements and bandwidth, it remains a subject of further study if a 2 Mbit/sec turbo decoder can be mapped on a FPGA, ASIC on the contrary has a high time

to market. When mapping to ASIC, a certain technology has to be specified. For the turbo decoder the CMOS18 technology is used, referring to a gate length of 0.18 μm. One of the outputs of a mapping tool is a static timing analysis. This analysis shows the delays of signals though the IC, using the wire model, and parameters specified. This delay can be used to determine the maximal clock frequency of that part of the IC. The netlist and a simple wire-load model can be used as an input for a power simulation tool. This tool is used for power estimation of the implementation.

5.2 High level and RT level turbo decoder design

The design of a flexible turbo decoder largely fits into high level and register transfer level design, explained in section 5.1.2. In that section 3 layers of processing where recognised: a flexible layer, a data flow layer, and a processing layer. Flexibility is required at each level. Flexibility exists at three levels:

- Design level (Des)
- Configuration time (Con)
- Run time (Run)

For implementation it is important to understand which flexibility is handled at which layer on which level. This flexibility is shown in table 5.2.

	flexible layer			data flow layer			execution layer		
variants | flexibility	Des	Con	Run	Des	Con	Run	Des	Con	Run
Max/Max⋆-log-MAP							*		
Dynamic scaling		*							
Normalisation technique							*		
programmable stopping		*							
Early stopping technique	*						*		
Early stopping criteria		*							*
NII/training	*			*					
vector compression	*		*						
vector reconstruction	*		*						
window length			*		*				
block length			*		*				
trellis termination	*								
Multi SISO modules	*								
Switching SISO 1/2			*						
Switching (de-)interleaving			*						

Table 5.2: flexibility

Note that configuration time flexibility of the data flow layer can be a run time flexibility of the flexible layer. The flexible layer may actually have to configure this data flow layer.

For the flexible (turbo architecture) layer, as expected, a lot of decisions have to be made on configuration time, and run time. This flexibility further reinforces the choice for a environment like A|RT Designer. For the data flow layer only two flexibilities are required: the window length and the block length. The choice for sliding window NII or sliding window with training calculations is a design choice. For the processing layer (the data paths) the early stopping criteria might be a low level time flexibility. Which early stopping technique is applied, which algorithm is executed, and which normalisation technique is applied is determined on design time.

5.2.1 Flexible turbo architecture

The flexible layer is the highest layer which can be recognised in the turbo decoder. This layer interfaces to the data flow layer. The most important aspects of choosing which functionality belongs in what layer, and how the interface is set up are determined by the data flow layer. It is necessary to give the data flow layer the most streaming based tasks and to prevent irregularities in this streaming. The solution to achieve this aim is a configurable SISO module. This configuration is in the form of parameters, which can be set for each processing block. The interface to this data flow layer is based on 3 items:

1. Each input output value should be transferred to the data flow layer only once. This is done to limit the bandwidth
2. Systematic and parity Log-Likelihood Ratios (LLR) are sent to the data flow layer. By using systematic instead of intrinsic and a-priori LLRs the bandwidth to the intrinsic systematic memory can be reduced by 50%
3. The output of the data flow layer is:
   - Soft Output
   - a flag, passed to the higher levels, to signal if the early stopping criteria for that bit is met. This could be a flag for each criteria, explained in section 4.5

For the Next Iteration Initialisation implementation technique, the memories for saving stakes, and memories for saving state metrics, part of the data flow SISO architecture, can be mapped onto one physical memory. Figure 5.7 shows how this single memory can be used for saving both stakes and state metrics.

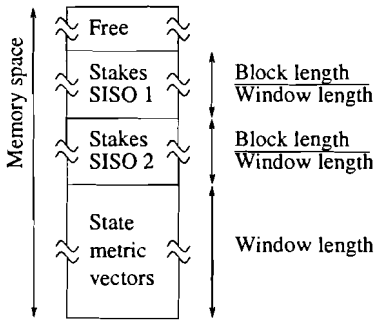

Figure 5.7: State metric vector memory for sliding window NII

The advantages of mapping the stake memory and the state metric memory on one physical memory are:

- Real flexible window length at run time
- Efficient memory (1 larger memory is more area efficient an better reusable by other functions)

For a (combined) compressed NII/NII the memory schedule might look as follows:
R_f stands for the reduction factor. The maximal reduction factor for 3GPP configurations, used in this thesis, equals 18. This is explained in section 3.2.3 and 4.4.4. Note that in a combined sliding window cNII/NII the compression is never applied if free memory is available. This is due to performance degradation of the cNII scheme. Section 5.2.2 further explains the design and implementation of the data flow layer.

The address generation is left out of the scope of this thesis. This is due to the specific interleaving algorithm for 3GPP, which was subject to further changes during the research project. For other projects, other interleaving algorithms can be used, requiring other architectures.

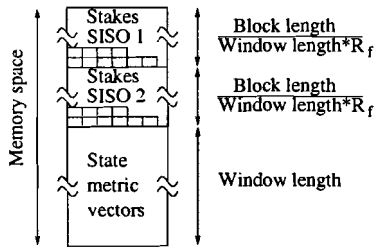

Figure 5.8: State metric vector memory for combined sliding window cNNI/NII

5.2.2 Data flow SISO architecture

The data flow layer is positioned at the second processing layer of the turbo decoder design. The interface to the higher layer is already discussed in section 5.2.1. This interface was largely determined by the use of PHIDEO. The interface between the data flow layer and the execution layer is also determined by the use of PHIDEO. This tool only provides data flow, memory and address generation. All processing has to be done in data path blocks. The implementation aspects of the data path blocks is explained in section 5.2.3.

During the design an implementation of the SISO decoder implementation decisions have to be made. As already explained, the Max($\star$)-log-MAP algorithms chosen. Second of all a decision for either sliding windows with training calculations or sliding window with NII has to be made. This last variant is actually implemented. The architecture however is open for both options.

Functional design

During the functional design, only the 'what' of the design is addressed. In principle the SISO unit has the next parameters:

- window length
- number of windows
- last window length

For the first version only the second parameter is implemented. No problems are expected for both a variable window length, and different last window length. Within each window a full forward recursion has to be done, followed by a full backward recursion.

The sliding window technique is applied, where the forward recursion is initialised in the beginning and the backward recursion is initialised every window. The basic schedule of the sliding window SISO is shown in Figure 5.9.

The Soft Output LLR have a different order compared to the inputs. This different order can be corrected by using a (slightly) more complex addressing mechanism for the (de-)interleaver, or by using an additional memory within the SISO unit. In this case the first solution is chosen. Note that stake 0 is sent to the output. This is done to avoid irregularities. The task schedule is shown in Figure C.3 in the appendix. In principle the algorithm is specified as a full block BCJR-type algorithm on a window length, where the forward state metric vector is passed to the next window and the backward state metric vector is retrieved from a higher level. This basic architecture can be used for all implementation techniques, explained in section 4.4. Note that for this schedule the forward recursion is the main recursion, and a continues stream of input and output is applied.

For implementing the sliding window with training calculations, the backward stakes do not have an interface to the flexible layer. One backward recursion unit is added. The schedule of this SISO unit is shown in Figure 5.10

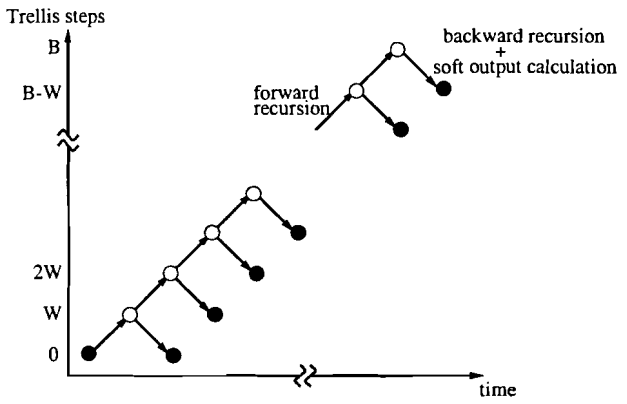

Figure 5.9: Basic sliding window schedule

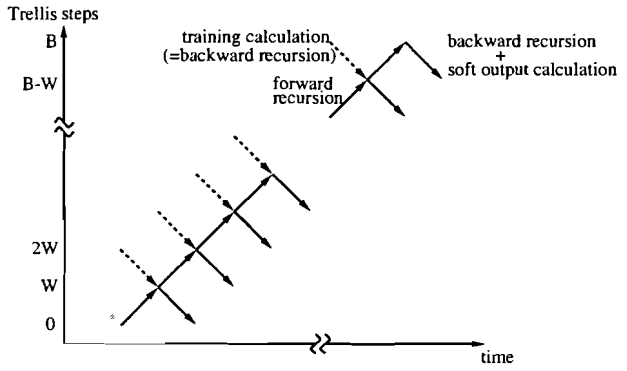

Figure 5.10: sliding window schedule with training calculations

Note that the first training calculations are done to reduce irregularities in the data flow. The actual calculations might be disabled to save power. For low block lengths this schedule gives more control overhead and less flexibility, compared to (compressed) NII, which can use the basic sliding window schedule.

Now that the 'what' of the design is explained, the 'how' is explained next.

Architectural design

The architectural design of the SISO unit is divided into 2 separate steps:

- data flow architecture
- memory scheduling and mapping

The data flow architecture is largely determined by the PHIDEO template. As explained in section 5.1.2, this architecture can be shaped by constraints. These constraints can be recognised in the pif description, shown in Figure C.2. These constraints lead to the architecture shown in Figure 5.11. Note that the backward stakes are not yet interfaced to the higher level. For the sliding window with training calculations the basic architecture will not change. A backward recursion unit, a memory file, and a feed back register, including multiplex operations, will be added.

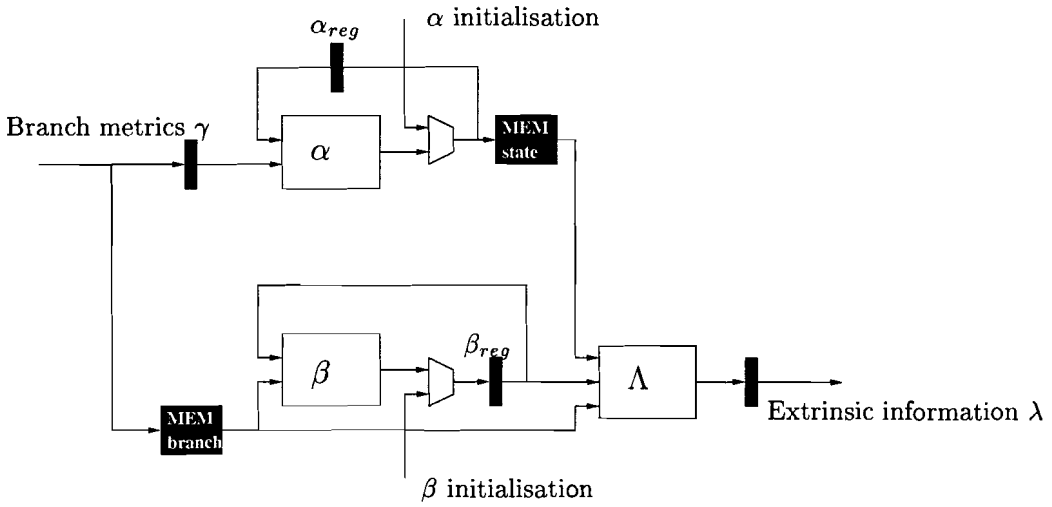

Figure 5.11: flexible SISO architecture

For memory two issues arise: mapping and scheduling. Smart memory scheduling results in very efficient memory usage. Figures C.4 and C.5 show the best possible memory schedule: 100 % utilisation. This memory schedule is applied for all memory files and can also be used for additional memory files (training calculations and same order Soft Input, Soft Output LLR). During the implementation the memory files where synthesised as register files. This is very inefficient for area cost. This area cost is further explained in section 5.4.1. Within PHIDEO it is possible to use external, more efficient, RAMs. By putting both the branch metrics, and the state metric vectors on the same address, address generation logic can be reduced and the efficiency of the SRAMs can be enlarged. This last advantage is the biggest due to the area and power cost division, explained in section 5.4.

5.2.3 Execution of data path operations architecture

The lowest processing layer has only an interface to the data flow layer. In the data flow layer, no data based computations are conducted. This is due to the set-up of PHIDEO. The functions: forward recursion, backward recursion, and Soft Output calculation, which can be recognised in the BCJR-type SISO decoder are designed manually using the VHDL language. This is due to the simplicity of these data blocks. The following implementation choices have been made:

- Max-log-MAP
- no additional hardware for early stopping criteria
- no delay optimisation or pipelining

These choices have no large impact and can be reversed without any problems. The data path of the forward recursion is shown in Figure D.1. The VHDL code is shown in Figure D.3. Note that normalisation is done by making α_0 always 0. No optimisations to remove subtractions are done. The data path of the Soft Output calculation is shown in Figure D.4. Note the add $\rightarrow$ Max $\rightarrow$ add $\rightarrow$ Max $\rightarrow$ Max $\rightarrow$ add calculation order. This order has high impact on the delay of the data path, as will be explained in section 5.4.3. The VHDL code is shown in Figure D.5.

These data paths are used to show the correctness of the SISO module. For a real product, data path optimisation has to be done, and overflow characteristics have to be implemented. For the sliding window with training calculations the backward recursion data-path is implemented twice, resulting in 4 data path blocks.

5.3 Verification

This section will explain the verification for both the C++ turbo decoder using YAPI and the VHDL model. This last step is an important one because for the C++ to VHDL path a different language (pif) has to be used. This excludes any form of correctness by construction. First the turbo decoder is specified in C++ and YAPI. The main reason for choosing YAPI is the fact that YAPI was used by the SCUBA project. An interface to this 'language' was required and functions like (de-)interleaving and turbo encoding were already available as YAPI processes. Verification of the turbo decoder is done by conducting simulations and comparing results to the results of others. Because the same BER performances are achieved the correctness of the code is assumed.

In earlier simulation results from H. Krauß, a performance degradation of the sliding window with training calculations is shown. Because of this performance degradation, alternatives were searched. EI-BCJR and sliding window NII were considered. Both alternatives are publiced by Dingninou et. al. [6] on a symposium in France. Later it was discovered that the code from H. Krauß, which he obtained from the 3GPP standard, contained bugs. These incorrect simulations had the next results:

- Further believe in the fact that incorrect code does not achieve the same BER performance
- Better verification of results from others is required
- An extensive search for implementation alternatives for the BCJR-type implementation was conducted

The extensive search for implementation alternatives was done using the YAPI environment. This environment is however not suitable for doing such an algorithmic analysis. Since YAPI is not used as an entry for A|RT Designer, PHIDEO, or data-path desciption, and the belief that it did not provide any new insights contribute to the meaning that the additional effort is not justifiable and that YAPI should not have been used for the turbo decoding work.

The working of the SISO unit, implemented using PHIDEO and VHDL, is checked by comparing the outputs of the YAPI processes with the output of a simulation tool, when the same input stimuli is used. This input stimuli is obtained by monitoring the YAPI fifos entering the SISO 1 processes during simulation. Since the output of both environments were exactly the same for more than 1700 input bits the two are considered equal.

For further work, both the C++/YAPI model and the data path blocks have to be made overflow resistant and the rounding techniques have to be applied and verified. A larger set of tests should be applied, including different block -, and window lengths and vector initialisation in the PHIDEO unit. It remains an open question if the YAPI environment should be used for these verifications or if the specification of the turbo decoder co-processor should be fully described in A|RT Designer, using its own testbench. BER performance can be compared to the YAPI performance shown in appendix A.

5.4 Implementation cost

This section shows the 3 main implementation cost: chip area, power dissipation and latency. The cost are obtained by synthesing the sliding window Max-log-MAP SISO module, which is designed, implemented, and verified. The implementation contains no test facilities and overhead expected from the actual routing. No effort is put into optimizing the solution for higher performance. The VHDL was simulated both at functional and architectural level. For the architectural level simulation, and the power sythesis a clock frequency of 1 MHz is used, without knowing the actual speed of the circuit. The speed of the circuit is largly determined by the delay of the data paths. Section 5.4.3 will further explain this delay.

5.4.1 Chip area

For analysing the chip area of a turbo decoder, the area cost of the SISO module and the memories are considered. The (de-)interleaver mechanism is left out of the scoop of this thesis, as explained in section 5.2.1. The Max-log-MAP sliding window SISO module has been designed and implemented using PHIDEO. The VHDL output of PHIDEO is synthesised. The output of a synthesis tool is an area report of the SISO module. This output is stated in the next table:

Module	Architecture name	Total Area (μm^2)
or_siso_low_rtl		436604.938
register_file_width64_addrwidth6_depth40_workspaces1_holdable0	MEM-state	263921.668
register_file_width7_addrwidth6_depth40_workspaces1_holdable0	MEM-branch	31121.408
register_file_width4_addrwidth6_depth40_workspaces1_holdable0	MEM-branch	18661.376
or_siso_low_l_pu_shell	Λ	26238.977
or_siso_low_a_pu_shell	α	21757.953
or_siso_low_b_pu_shell	β	20860.929
or_siso_low_controller		26603.521
or_siso_low_address_generators		6795.264
rege_ww64_0	α_{reg}	4980.736
rege_ww64_1	β_{reg}	4980.736

The architecture name is confirm the naming in Figure 5.11. The chip area of the local memories (implemented as register files) is dominant in the architecture. Since all 3 memories have equal schedules it is possible to combine them into 1 wide word memory (all three words, mapped on one word of the new memory). This wide word memory can be implemented using SRAMs. Implementation cost for such an SRAM in c050 technology are given in the next table:

```
|-------------------------------------------------------------------------------|
|      AMDC QSRAM ESTIMATOR                    QSRAM  : 1.0                     |
|      --------------------                    Date   : 26/04/2000              |
|-------------------------------------------------------------------------------|
| Configuration                | bwe |rwmode|ceconf| daioco | Special demands :     |
| input: w=   40 b= 75         |-----|------|------|--------|                       |
| x*y*z: number of words       |   0 |   RW | PSYNC|  NOREG | system freq =   1 MHz |
|-------------------------------------------------------------------------------|
| Size |               | Xdim | Ydim | Area |  Eff | tp_c_q | tcyc |       | power |
| bits |   x   y   z   b |  mm  |  mm  |  mm2 |   %  |   ns   |  ns  |       | mW/MHz|
|------|-----------------|------|------|------|------|--------|------|-------|-------|
|  2.9K|  20   2   1  75 | 0.61 | 0.25 | 0.16 |  28  |   3.1  |  3.4 |       |  0.39 |
|  2.9K|  10   4   1  75 | 1.10 | 0.19 | 0.21 |  21  |   3.2  |  3.9 |       |  0.51 |
|-------------------------------------------------------------------------------|
```

Note that a word width of 75 bit is used: 64 bit for saving the eight 8 bit state metrics, 7 bits for the systematic branch metric and 4 bit for the parity branch metric. Since the PHIDEO unit is synthesised for c018 technology, area cost for SRAM's have to be converted to this technology. This conversion can be done using a correction factor s^2, where s descibes the geometric effects and equals approximately 0.7. For the first memory option in the table, area costs then are 76250 μm^2. This is a area reduction of a factor $\frac{76250}{313703} \approx \frac{1}{4}$. This wide word memory is used for the next analyse.

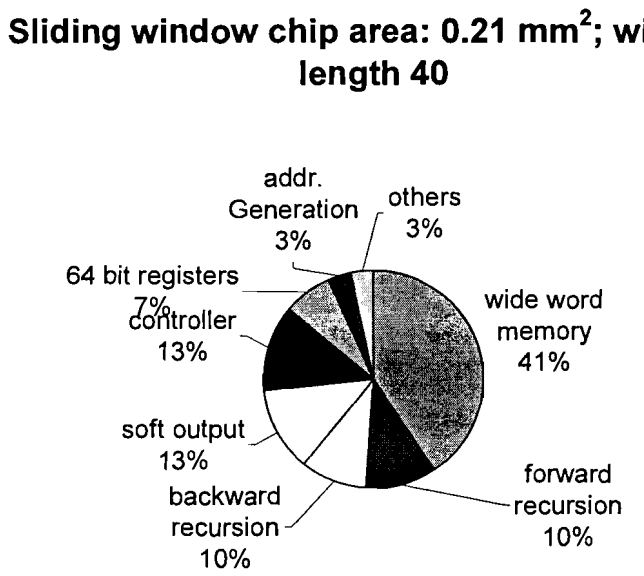

Figure 5.12: Area cost division of the SISO decoder

Figure 5.12 shows the area cost division for the SISO module. Note that, although memory optimization is done, memory is still dominant in chip area. This is the area cost of the Max-log-MAP sliding window SISO decoder. Initialisation is done with the uniform state metric vector. For this area pie the window length is chosen 40. The implementation cost of the sliding windows with training calculations and the sliding window Next Iteration Initialisation is shown in Figure 5.13. Sliding window with training calculation has the same logic area cost compared to EI-BCJR (two backward recursion units), and a constant memory area cost of window length 40. Sliding window NII has lower logic area cost compared to EI-BCJR (one backward recusion unit), and a variable window length. The minimal number of addresses for saving stakes equals twice the window length, against one times the window length for EI-BCJR. This window length is the square root of the block length. The memory requirements for temporary storage and state metric storage grow also with the window length. The cost are explained in section 4.4. Figure 5.13 also shows the area cost for intrinsic and a-priori LLRs. The area for these memories grow linear with the block length and can not be avoided. Note that for block lengths larger than 1200 these memories become dominant.

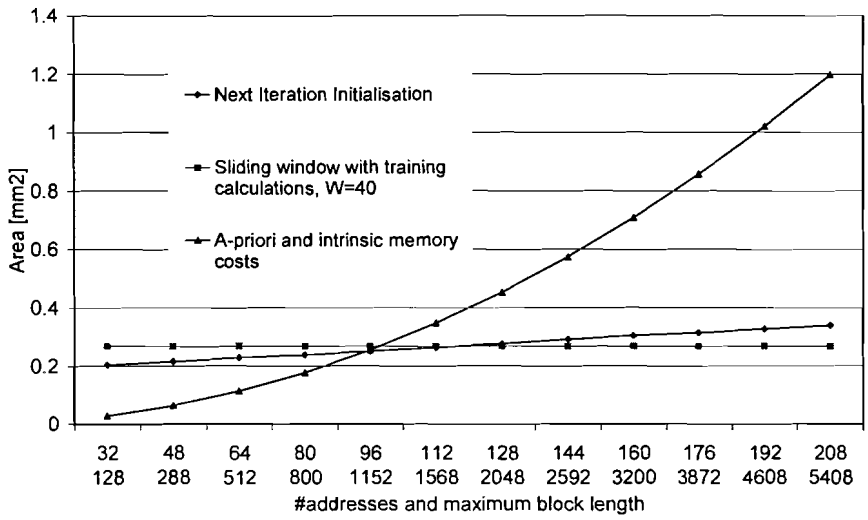

Figure 5.13: Area cost of the turbo decoder

5.4.2 Power dissipation

An other important implementation cost figure for mobile communication is power. Power figures can be obtained using synthesis tools. The next table shows the output of such a tool:

```
****** Diesel instantiation information ******
Diesel version:                 2.1.1
Build for:                      NC-VERILOG v2.2.s13
Simulation date:                Wed Apr 26 13:49:49 2000

Capacitance function:           (NLOADS<>0.000e+00)?(NLOADS*8.000e-15):6.000e-15;

Wire cap. includes fanin:       YES
Delay back annotation:          NO

Process name:                   CMOS18
Char. temperature:              25.00  [Celsius]
Char. supply voltage:           1.800  [Volts]
Number of library cells:        61

Default slope rise:             300.0p [seconds]
Default slope fall:             200.0p [seconds]

Monitored design cell:          or_siso_low_abstract
Total number of design cells:   14865


Evaluation time interval (1):       0 , 1.766m [seconds]

netid  CGND  CWIRE  SRISE  SFALL  #1 #d #0-1 #1-0 #other energy net-name
----- ------ ------ ------ ------ -- -- ---- ---- ------- ------ ---------
   76      0  6.000f 300.0p 200.0p  0  0 1766 1766       1 478.7n :dut.ck


****** Hierarchical energy information ******
```

```
Evaluation time interval (1):       0 , 1.766m [seconds]

tot.energy  av.power hierarchical block name
---------- --------- ----------------------
   308.48n  174.63u :dut
   308.48n  174.63u :dut.dut
     3.38n    1.91u :dut.dut.address_generatorsLDHV
     6.44n    3.64u :dut.dut.controllerLDHV
     4.28n    2.42u :dut.dut.Sys_mem_regfileLDHV
     2.73n    1.55u :dut.dut.Par_mem_regfileLDHV
    31.83n   18.02u :dut.dut.A_mem_regfileLDHV
     9.30n    5.27u :dut.dut.Ab_mem_regLDHV
    10.53n    5.96u :dut.dut.B_mem_regLDHV
   106.78n   60.45u :dut.dut.or_siso_low_l_pu_shellLDHV
    52.16n   29.53u :dut.dut.or_siso_low_a_pu_shellLDHV
    60.42n   34.20u :dut.dut.or_siso_low_b_pu_shellLDHV
```

From the monitored net information information about the clock dissipation can be derived. This dissipation is $\frac{478.7}{787.24} \approx 60$ % of the total dissipation. The dissipation in the cells is shown in Figure 5.14.

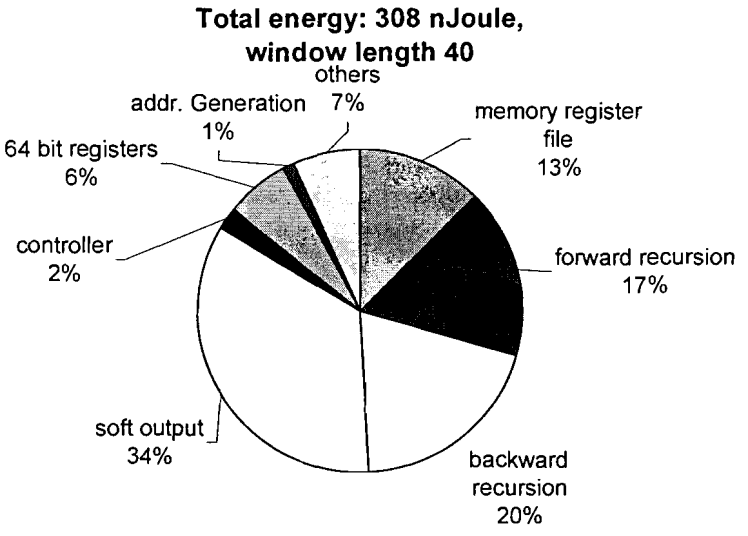

Figure 5.14: Power cost division of the SISO decoder

Note that the window length for this power pie is set to 40. 1760 bits are evaluated (1 MHz clock). This results in $\frac{787.24}{1760} \approx 0.447 \frac{nJoule}{bit}$. For a 2 Mbit/sec 10 iteration turbo decoder, power dissipation sums up to 17 mWatt. Note that register files are used as memory, in contradiction to the area estimation where SRAMs are used. For SRAM CMOS18 technologie we can scale power dissipation with a factor $s \cdot p^2$, where s discribes the geometric effects and p the reduction in supply voltage [22]. Both parameters are approximately 0.7, resulting in a scaling factor of 0.35. In the table of the 'AMDC QSRAM ESTIMATOR' in the previous section, the energy figure of 0.39 $\frac{nJoule}{access}$ is given. For every bit a read and write access has to be done. Combined with the scaling factor, the 40 addresses SRAM will dissipate approximately $0.27 \frac{nJoule}{bit}$, $\frac{0.27}{0.447 \cdot 0.13} = 4.66$ times larger than power dissipation of the register files. The figures for SRAM are worst case figures, while the figures for the register files do not correct clock dissipation (clock dissipation in register file is expected to be higher than average) and have no test facilities. These 3 items will assure that on average the power dissipation of the two memory variants is less than a factor 3.

5.4.3 Delay

The maximum clock frequency of a circuit is, in many cases, determined by the delay of the longest path. The longest path consists of memory access times, wire delay, and logic delay. The memory delay is given by the techology parameter. If a SRAM is used, which has a physical location further away from the logic, the wire delay of the SRAM to the logic has to be considered. Deep sub micron effect emphasis the importants of wire delay, even between logic. In this analysis the wire delay is neglected. The logic delay can be calculated easily. Starting after a clock rise, a signal propagates through a circuit, receiving a delay penalty for passing each logic function. When two values are added, the longest delay path is the path passing the carries of the full/half adders in de addition. This path is called the carry ripple path. If two four bit values are added the longest path consists of 4 carry ripples. If three 4 bit values are added, the carry ripple path extents with 1 carry ripple (more ripples are possible, depending on the quantization scheme). In general, concatenation of additions/subtractions do not introduce significant additional delay. This is shown in the left example of Figure 5.15. On the other hand, a maximum operation followed by an addition, does introduce a significant additional delay. Due to the multiplexers in the maximum operation the carry ripple restarts at the beginning of the additions. This effect, further refered to as Max → add, is shown in the right example of Figure 5.15.

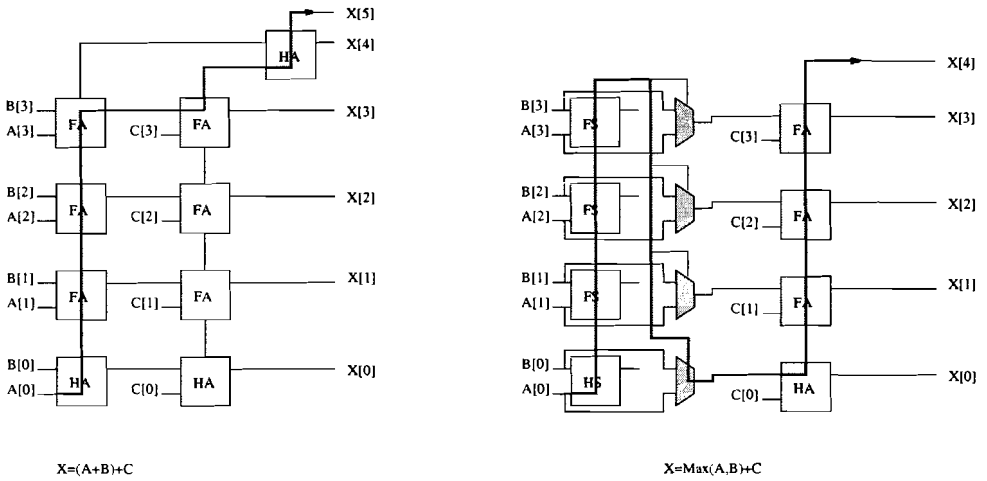

Figure 5.15: Delay model

Two techniques to reduce the longest path are: flow graph transformation, and pipelining. During flow graph transformation, operations are repositioned using mathematical transformation, for example $Max(A,B)+C = Max(A+C,B+C)$. In this case it will result in more operations. This technique is used in Figure D.2 on page 96 During pipelining, registers are shifted into the data path, resulting in shorter delays. The implications are that the latency of the operations becomes more than one clock cycle, while the data introduction interval stays one clock cycle.

The logic delay of the SISO unit implementation has been analysed using a synthesis tool. The longest path is found in the Soft Output calculation unit. The data-path of this calculation is shown in Figure D.4. The delay is mainly caused by the Max → add → Max → Max → add construction. The total delay of this path is 20 nSeconds. Note that the analysis is a worst case analysis, no wire delay is considered, additional delays for SRAM usage is not accounted, and the implementation is not optimize for delay. The delay of this Soft Output calculation can easily be reduced using a combination of flow graph transformation and pipelining. The delay of the forward and backward calculations is approximately 12 nSeconds. If timing constraints, pipelining (in the Soft Output unit), and flow graph transformations are applied, a 100 MHz turbo decoder can be achieved easily.

With further effort (amoungst others: carry look a head technology), clock frequencies of 250-300 MHz can be reached, which is as fast as the SRAMs.

An additional advantage of low path delay is power. Reducing the path delay also reduced the glidges, which are largely responsible for power consumption. This effect can already be seen in Figure 5.14. Although the soft output calculation consists of approximately 28% more operations compated to the recursive operations, power consumtion is 70% higher.

Chapter 6

Design space exploration

The algorithm complexity explained in chapter 3, the implementation aspects shown in chapter 4, and the experience obtained during the implementation exercise in chapter 5 are now used as input for the design space exploration. In the original design space the performance is positioned against the cost. The design space is complicated by the large variety of implementation variants and possibilities to exchange cost. The performance is therefore not being positioned against one unique cost figure but against several implementation alternatives with different cost figures. In many cases the design space exploration is not a discussion between performance and cost but more a discussion between cost exchange itself. The cost of a turbo decoder implementation can be divided into the next elements:

- Chip area
  - logic
  - memory
- Power dissipation
- Timing
  - latency (time shape in number of clock periods)
  - delay (clock speed in nano-Seconds)
- Bandwidth

For many implementation aspects/techniques these different cost are exchangeable. Imagine for example 2 cases:

1. The turbo decoder has the same speed as the data I/O
2. The turbo decoder is twice as fast as the data I/O

In the first case two input memories are required, resulting in a chip area of approximately 1.6 mm^2. In the second case $1\frac{1}{2}$ input memories are required, which uses a chip area of approximately 1.2 mm^2. Note that the gain is larger than the total area cost of the SISO module. This is exchange of latency and memory is explained in section 5.1.1.

During the discussion so far, some elements for making decisions are neglected. These remaining elements in the design space are more subtle, difficult or even impossible to measure/quantise. The following elements can be recognised:

- Patent avoidance
- Re-usability (change of focus)
- Flexibility (block length adaption)
- Implementation time

In the implementation stream many implementation aspects/options can be recognised. These aspects/options are divided into 4 sub-streams. In each sub-stream a decision tree is set up. For each alternative the design space of the alternatives have to be taken into account. Depending on the importance of each dimension, a decision can be made.

The **first sub-stream** is the algorithmic discussion, shown in figure 6.1. The questions are stated in hexagons, while the options are stated in the rectangles. The decision trade-off is discussed from left to right. The first discussion is the one about which decoding family has to be chosen. For a more detailed discussion refer to section 3.3.

When working under high hardware/area constraint, modified SOVA will be a very good alternative. When requiring high performance BCJR will be the best alternative. In most cases however, a design space has to be explored and, combined with early stopping, BCJR is the better algorithm family.

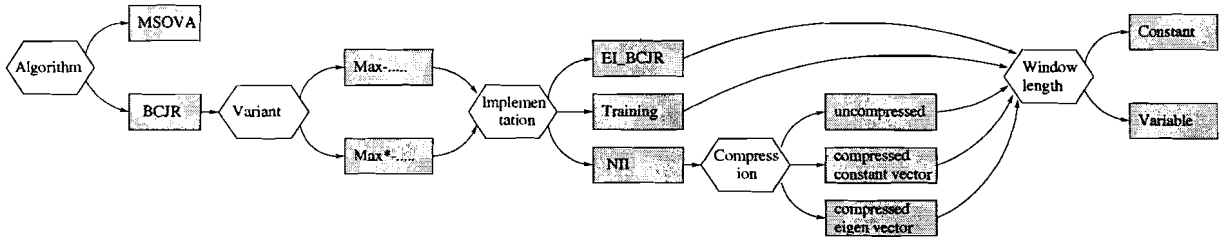

Figure 6.1: Algorithm decision tree

When choosing for the BCJR algorithm, the second choice is the choice between Max-log-MAP and Max$\star$-log-MAP. The implementation discussion between the two variants is held in section 4.2. The (wide) performance gap between the two can be reduced by scaling the extrinsic information. The influence of scaling extrinsic information on the BER/WER is explained in appendix A.5. The implementation advantages of using the Max operation instead of the Max$\star$ operation are: logic area (30 or 38 extra addition for Max$\star$ operations) and delay expensive Max $\rightarrow$ add data flows (discussed in section 5.4.3). Note that the logic area increases with approximately 30 % due to the extra additions (even without the logic required for implementing the correction function).

The third question is which of the 3 feasible implementation variants is chosen. These implementation variants are explained in section 4.4. The first alternative (EI-BCJR) has only theoretical interest. It is a true BCJR algorithm with optimal performance (for decoding a convolutional code). The second alternative is sliding window with training calculation. Performance differences between this sub-BCJR algorithm and EI-BCJR are negligible. The latency however is half, and the state metric memory is $\frac{40}{2 \cdot 72}$ compared to EI-BCJR. The last alternative is NII. The BER/WER performance is discussed in appendix A.3. For higher iteration numbers (and/or higher window lengths) performance differences between EI-BCJR and NII are

negligible. This implies that, on average, more iterations have to be done. Estimations differ between $\frac{1}{4}$ and approximately 0 additional iterations. NII is more memory intensive, while the training calculation variant is more computational expensive (30 % more operations). In principle NII uses less power but the additional iterations might disturb this conclusion. The advantages of the memory intensive NII are:

- Flexibility due to extention of the variable window length (using the same memory for state metric vectors and stakes)
- High reuse possibility and low implementation time of memory
- Full block BCJR for low block lengths for which BER/WER performance is already bad
- Avoiding the patent on sliding window with training calculations (patent number 5,933,462 Aug. 3, 1999)

The disadvantages are the higher memory cost and the average higher number of iterations. The area trade-off is shown in Figure 5.13. An other disadvantage of NII is that the stake memory cost for a 16 state RSC are doubled, resulting in an other area trade-off.

When making the choice for NII a fourth question is whether to apply vector compression and, if applied, which reconstruction mechanism is used. The BER/WER performance influence is shown in appendix A.4. A critical note for the vector compression usage is the fact that the input and a-priori memories grow linear with the block length, while the stake memory grows order square root with the block length. The 'lower' BER/WER performance of compression might result in higher iteration numbers, which result in higher latency, which might result in larger input memories. These addition input memories might consume more area than the reduction due to compression. The difference in BER/WER performance is expected to reduce due to better vector reconstruction schemes and values. In a variable environment with variable block lengths a combined NII/ compressed NII scheme can be used to choose an optimal (more independent) memory space. Large blocks then could use compression for some stakes to fit into the memory. For a more detailed discussion, refer to section 5.2.1. Note that the compression and reconstruction will require some additional hardware/software.

This last discussion also has a strong link to the last question in the algorithmic decision tree: the window length. For EI-BCJR a window length as large as possible has to be chosen. The window length is only restricted by $W + \frac{B}{W} \leq M$, where M is the number of memory addresses. For sliding window with training calculations the window length is determined by the number of training calculations which have to be conducted. For Rayleigh fading channel $10 \cdot (\mu+1)$ (=40) training calculations result in negligible performance loss. Due to additional complexity in the controller and branch metric storage it is wise to choose the window length larger or equal to, preferable a multiple of, the number of training calculation. For the NII variant it is assumed that the state metric vector and stakes are in one memory. This memory has to have between 65 and 216 entries for the 3GPP implementation. The window length is determined by Latency (B+W) and performance, and is restricted by $W + 2 \cdot \frac{B}{W \cdot R_f} \leq M$. For a more detailed discussion, refer to section 5.2.1

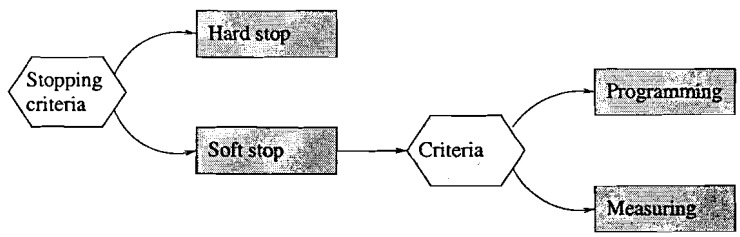

Figure 6.2: Stopping decision tree

The **second sub-stream** is expressed in the stopping decision tree. This tree is shown in Figure 6.2. If an application has high latency constraints (maximal delay is ... μSeconds) and that delay is smaller or equal to the time required to compute the average number of iteration, a hard stop might be required. Note that this will influence the BER/WER performance. At small implementation cost it is possible to make the stopping criteria programmable (soft stop). Note that the soft stop does not exclude a constant stopping criteria. The advantages of soft stop are: power dissipation, average latency, and flexibility. It provides the higher layer a handle for quality of service aspects. Within the soft stop variant there are 2 variants: programming and measuring. In the programming variant the higher layer determines if the turbo decoder has to stop iterating. Reasons for stopping could be latency, expectation of correctness, and/or believe in lack of further improvement. The measuring variant, known under the name early stopping, is discussed in section 4.5. At very low logic area cost (or higher memory area) it is possible to estimate if the block, which is currently under processing, is correct. This technique is alway used in combination with a maximum number of iterations. Note that for early stopping additional input memories might be required. For programmed stopping the controller might prevent this additional memory.

The **third sub-stream** is expressed in the quantization decision tree. Quantization is explained in section 4.1. Although more questions are possible, the two main questions are shown in the decision tree. Note that it is not a real tree. First the question of word width is addresses. After having chosen an option, normalisation is addressed. A decision for normalisation is best done after knowing the word width because of the (small) impact of word length to normalisation. Note that normalisation is a quantization technique for a vector.

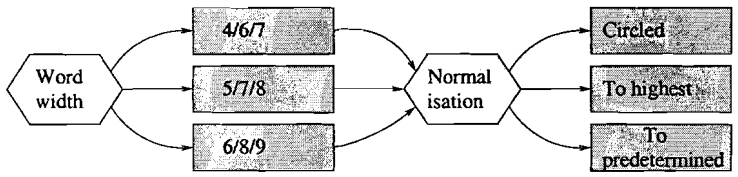

Figure 6.3: Quantization decision tree

The discussion on word width is first of all an area versus BER/WER performance discussion. Area scales approximately linear with the word width. Larger word widths might change the area crosspoint between NII and sliding window with training calculation and enlarge the compression factor for compressed NII. At the third place the (minor) additional delay is addressed. This additional delay might actually prevent some high clock frequencies. On the other hand leads higher performance, when using early stopping, to lower iteration numbers, which leads to lower latency. After making the decision on the word width, the second question is normalisation. This subject is discussed in section 4.3. The first option is the circled technique. This technique has very low latency but high memory cost. The second technique is to find the highest state metric and to subtracted it from each state metric. Finding the highest state metric has high delay cost. Logic area is slightly higher but negligible compared to the reduction in memory area. For each state metric saved, 1 bit less has to be saved. The last technique is to subtract a predetermined state metric from each state metric. This technique has lower (or equal) memory requirements compared to the previous techniques for the 8-state code and the options for word width. Latency is slightly higher than the circled technique but this can be solved with flow graph transformation, explained in section 4.3.

The **fourth sub-stream** is the multiple resources dimension. This discussion starts at the point were a single turbo decoder with one SISO module is not fast enough. These speed requirements can be determined by latency, input memory resources, and throughput. Before looking at multiple turbo decoders, first multiple SISO modules have to be considered. Figure 6.4 shows the decision tree that is used.

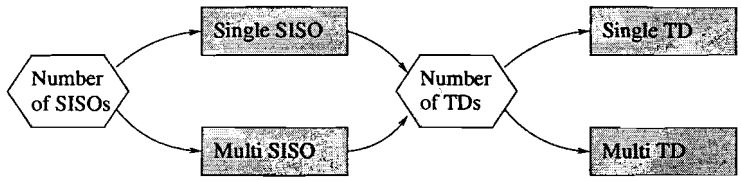

Figure 6.4: Multi resources decision tree

By adding a SISO unit it is meant that the SISO unit is going to process the same block. Due to the influence of the window length on the performance, it is only interesting to use multiple SISO units if the block length divided by the number of SISO units results in a large enough (multiple of) the window length. Two SISO modules on a block length of 60 bits is no good idea. The extension of SISO modules is also limited by the bandwidth to the input and a-priori memories. If the turbo decoder has to process more data than one turbo decoder can handle, multiple turbo decoder issues have to be discussed. This multi turbo decoder discussion is done in the concept phase of the design, explained in section 5.1.1.

Chapter 7

Conclusions and Recommendation

It is the aim of this thesis to show the implementation of a turbo decoder and to explore the design space of performance and cost. The general conclusions of this thesis are:

- The turbo decoder can be implemented using the tools A|RT designer, PHIDEO, and the VHDL language for the three layers of processing: flexible, data flow, and execution respectively
- Area cost and power dissipation of the Soft Input/Soft Output (SISO) unit, the most complex unit inside the turbo decoder, are low compared to the total chip budget (1 cm^2, 1 Watt)
- Conclusion for the design space involving performance:
  - BCJR type algorithms are more flexible and have higher performance compared to Viterbi type algorithms
  - The performance differences between Max$\star$-log-MAP and Max-log-MAP can be reduced, using scaling of extrinsic information
  - The memory requirements of sliding window Next Iteration Initialisation can be reduced, using vector compression, at the price of performance
- Conclusions for the design space involving cost:
  - Normalisation by subtracting a predetermined state metric is the best alternative for 3GPP configurations
  - The computational cost of Max$\star$-log-MAP is approximately 30% higher, compared to Max-log-MAP, resulting in larger logic area and higher delay
  - Sliding window Next Iteration Initialisation implementation technique is more memory intensive, while sliding window with training calculations is more logic intensive
  - NII is the most flexible implementation technique of the BCJR type algorithms
  - The SISO unit can decode a block of data in approximately B clock cycles, where B is the block length of the data block
  - The clock frequency of the SISO unit can easily run at 100 MHz. Clock frequencies of approximately 250-300 MHz can be reached using logic circuitry optimisations
  - Multi SISO, multi Turbo decoder architectures can be used to achieve very high speed turbo decoders

The recommendations for further work are divided into two items:

- For the turbo decoder in general
  - Use the A|RT designer environment to implement a flexible turbo decoder
  - Analyse performance loss and implementation cost of quantization
  - Analyse the effects of different input quantization schemes
  - Analyse the scaling factors and scaling schemes for scaling extrinsic information
  - Analyse the techniques (eigen vector and constant vector), factors and schemes for vector reconstruction
- For the SISO in particular
  - Implement the variable window length and different last window length
  - Use SRAMs in stead-off of register files for memory requirements
  - Optimise the data paths

Bibliography

[1] L. R. Bahl, J. Cocke, F. Jelinek, and J. Raviv. Optimal decoding of linear codes for minimizing symbol error rate. *IEEE Transactions on Information Theory*, IT-20:284–287, March 1974.

[2] C. Berrou, P. Adde, E. Angui, and S. Faudeil. A low complexity soft-output viterbi decoder architecture. In *IEEE Proceedings of ICC '93*, pages 737–740, Geneva, May 1993.

[3] C. Berrou, A. Glavieux, and P Thitimajshima. Near shannon limit error-correcting coding and decoding: Turbo codes. In *IEEE Proceedings of ICC '93*, pages 1064–1070, May 1993.

[4] S. Crozier, Ken Gracie, and Andrew Hunt. Efficient turbo decoding techniques. Technical report, Communications research Centre, 3701 Carling Avenue, P.O. Box 11490, Station H Ottawa, Ontario, 1999.

[5] E.A. de Kock, G. Essink, W.J.M. Smits, P. van der Wolf, J.-Y. Brunel, W.M. Kruijtzer, P. Lieverse, and K.A. Vissers. Yapi: Application modeling for signal processing systems. In *Proceedings 37th Design Automation Conference*, June 2000.

[6] A. Dingninou, F. Raouafi, and C. Berrou. Organisation de la memoire dans un turbo decodeur utilisant l'algorithm SUB-MAP. In *Proceedings of Gretsi*, pages 71–74, September 1999. France.

[7] M. P. C. Fossorier, F. Bukert, S. Lin, and J. Hagenauer. On the equivalence between SOVA and Max-log-MAP decodings. *IEEE Communications Letters*, 2(5), May 1998.

[8] W. J. Gross and P. G. Gulak. Simplified MAP algorithm suitable for implementation of turbo decoders. *IEEE Electronics Letters*, 34(16):1577–1578, August 1998.

[9] J. Hagenauer and L. Papke. Decoding 'turbo'-codes with the soft output viterbi algorithm (SOVA). In *Proceedings of IEEE International Symposium on Information Theory (ISIT'94)*, page 164, Trondheim, Norway, June 1994.

[10] J. Hagenauer, P. Robertson, and L. Papke. Iterative (turbo) decoding of systematic convolutional codes with the MAP and SOVA algorithms. *ITG Tagung, Codierung fur Quelle, Kanal und Ubertragung*, pages 21–29, October 1994.

[11] Goo hyum Park, Suk hyun Yoon, Ik soo Jin, and Chang eon Kang. A block-wise MAP decoder using a probability ratio for branch metrics. In *IEEE 50th Vehicular Technology Conference*, volume 3, pages 1610–1614, September 1999.

[12] Gibong Jeong and Dan Hsia. Optimal quantization for soft-decision turbo decoder. In *IEEE proceedings of 50th Vehicular Technology Conference*, volume 3, pages 1620–1624, September 1999.

[13] C. G. Clark Jr and J.B. Cain. *Error correction coding for digital communications*. New York: Plenum Press, 1981.

[14] H. Krauß. Complexity of a Max*-log MAP decoder for use in a UMTS turbo decoder. 0.3d, Philips Semiconductors TCMC Nurnberg, July 1999.

[15] H. Krauß. Turbo decoder simulation results for AWGN. analysis report s25. 0.2d, Philips Semiconductors TCMC Nurnberg, March 2000.

[16] G. Masera, G. Piccinini, M. Ruo Roch, and M. Zamboni. VLSI architectures for turbo codes. *IEEE Transactions on VLSI Systems*, 7(3):369–378, September 1999.

[17] A.K. Nieuwland and P.E.R. Lippens. A heterogeneous hw-sw architecture for hand-held multimedia terminals. In *IEEE Workshop on Signal Processing Systems, SIPS'98.*, pages 113–122, October 1998.

[18] P. Popescu. Complexity analysis of the m-SOVA algorithm. Technical report, Philips Consumer Communications, July 1999.

[19] P. Robertson, E. Villebrun, and P. Hoeher. A comparison of optimal and suboptimal MAP decoding algoritms operating in the log domain. In *Proceedings 1995 International Conference on Communications*, pages 1009–1013, 1995.

[20] C. E. Shannon. A mathematical theory of communications. *Bell System Technical Journal*, 27((3,4)):379–423, 623–656, 1948.

[21] K. van Eijk, B. Mesman, C. A. Alba Pinto, Qin Zhao, M. Bekooij, J. van Meerbergen, and J. Jess. Constraint analysis for code generation: Basic techniques and applications in facts. In *ACM Transactions on Design Automation of Electronic Systems (submitted)*, October 2000.

[22] J.L. van Meerbergen. Embedded multimedia systems in silicium. Technische Universiteit Eindhoven, 1999.

[23] J.L. van Meerbergen, P.E.R. Lippens, W.F.J. Verhaegh, and A. van der Werf. Phideo: High-level synthesis for high throughput applications. *Journal of VLSI Signal Processing*, 9:89–104, 1995.

[24] Z. Wang, H. Suzuki, and K. K. Parhi. VLSI implementation issues of turbo decoder design for wireless applications. In *IEEE Workshop on Signal Processing Systems*, pages 503–512, 1999.

Appendix A

Simultion results

A.1 Simulation Framework

The following simulation framework is used in this appendix:

Turbo Encoder

- 8-state PCCC (memory $\mu = 3, N = 2^{\mu} = 8$), generator polynomial $G(D) = \left[1, \frac{1+D^1+D^3}{1+D^2+D^3}\right]$
- rate $\frac{1}{3}$
- prime interleaver of 3GPP: block-length B (equivalent to one frame of input data)
- conventional trellis termination with 6 tail bits (3 systematic and 3 parity bits), i.e. $B_{Tail} = 6$
- output block-length: $3 \cdot B + 2 \cdot B_{Tail}$, including $B + B_{Tail}$ systematic bits

Channel model

- AWGN
- ideal soft values are assumed

Turbo Decoder

- floating point accuracy
- soft input values are not quantized
- Max⋆-operation: $\max(x) + f_c(|x - y|)$ with tabulated correction function $f_c(x)$, 64 entries linear distributed on the interval [0, 0.1, ..., 6.3]
- maximum number of iterations: 10
- criteria for terminating the simulations: 100 word errors

A.2 Reference simulations

Figure A.1 shows the performance of the full block Max⋆-log-MAP decoding scheme in case of a block-length of 320 bits for up to 10 iterations. This serves as the reference for further comparison with other BCJR-type implementation techniques. After one iteration the largest performance gain is achieved. In each following iteration the performance gain decreases. After 6 iterations the performance gain becomes very small, implying that for practical implementations a large number of iterations is not cost-effective

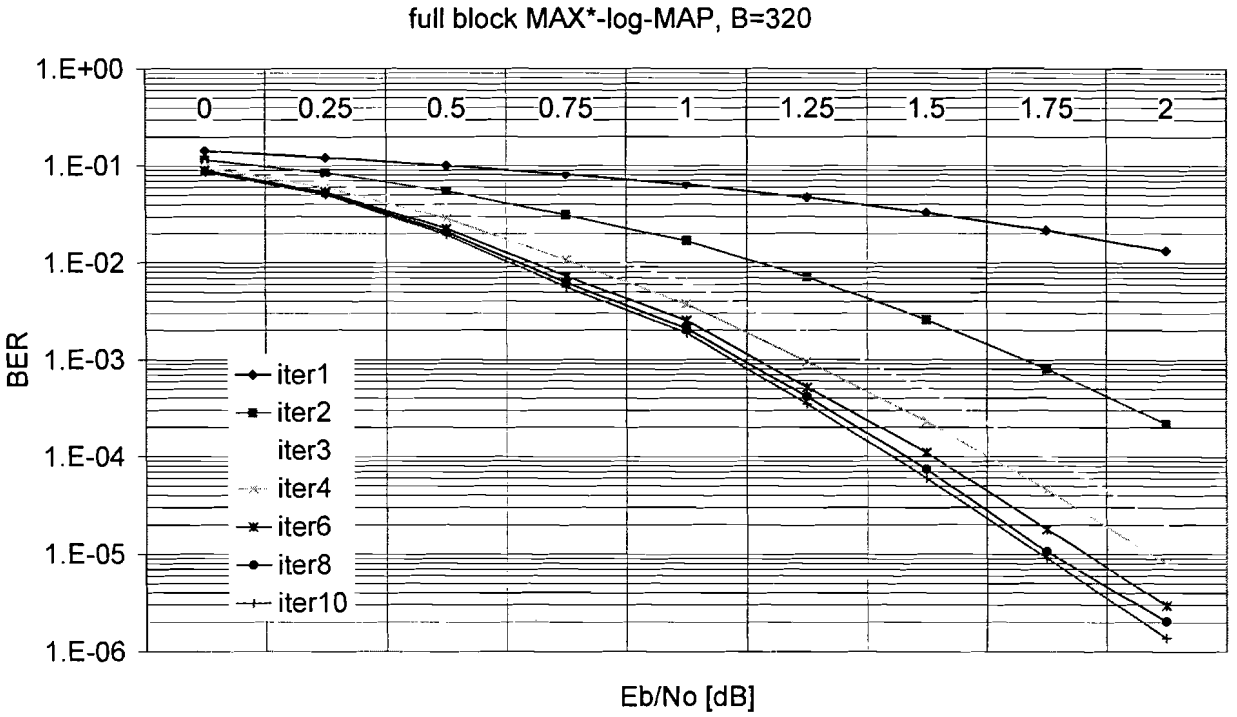

Figure A.1: BER versus Eb/No for different iterations

In our simulations we compare bit error rates (BER) and word error rates (WER) versus Eb/No. Eb/No is the energy per bit divided by the noise energy.

Figure A.2 shows that the BER performance improves if larger block-lengths are applied. This effect is called interleaver gain and is the result of better interleaving due to the larger number of bits.
A SISO decoder expects uncorrelated soft input. If no interleaver is used, the soft inputs are correlated. An error in the trellis of RSC 1 would lead to an error at the same place in RSC 2. It is the aim of an interleaver to uncorrelate the information of RSC 1 and RSC 2. Larger interleavers succeed better, because of the larger space they have.

A.3 Effects of window initialisation

Due to memory cost it is expensive to calculate the Max(⋆)-log-MAP algorithm on the entire block. The alternative implementation technique is sliding window Max(⋆)-log-MAP. Several initialisation schemes can be applied. Figure A.3 shows the results without an initialisation, meaning that the backward recursion at stakes W, 2W, ..., B is initialised with uniform state metric vector (e.g. all zero). After 2 iterations, BER performances are far less compared to full block Max⋆-log-MAP. At a BER of $3 * 10^{-4}$ full block Max⋆-log-MAP performs more than 1.3 dB better.

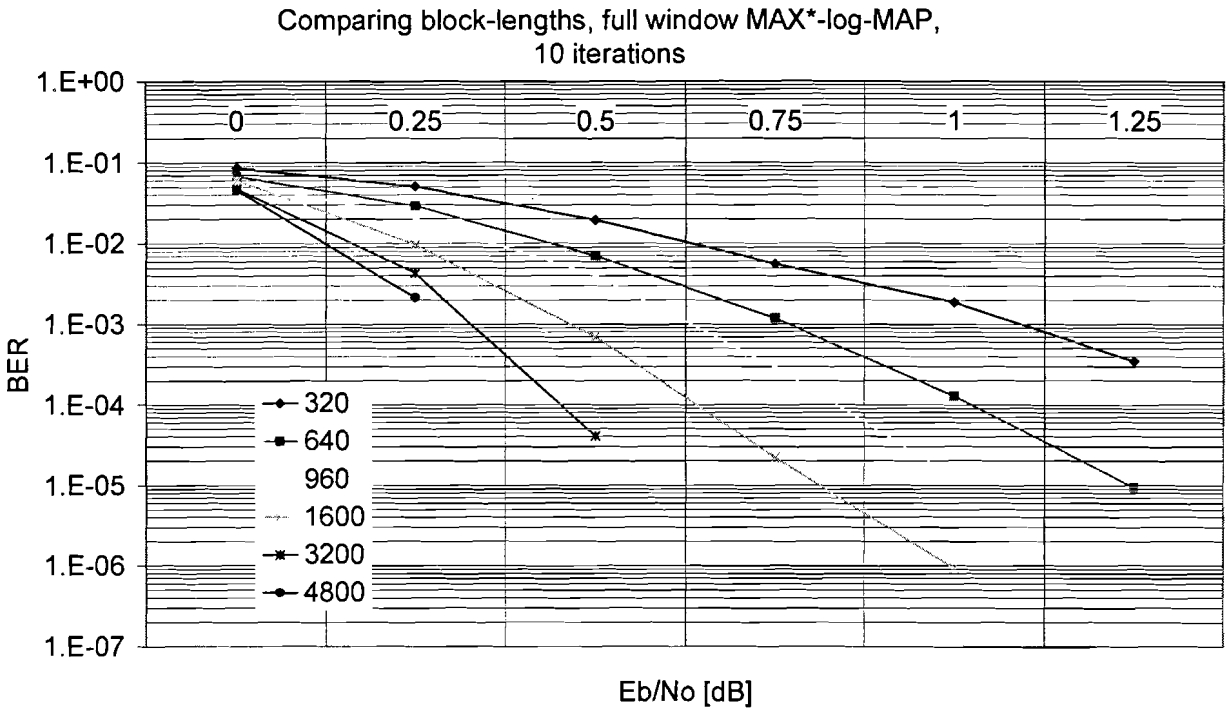

Figure A.2: BER versus Eb/No for different block-lengths

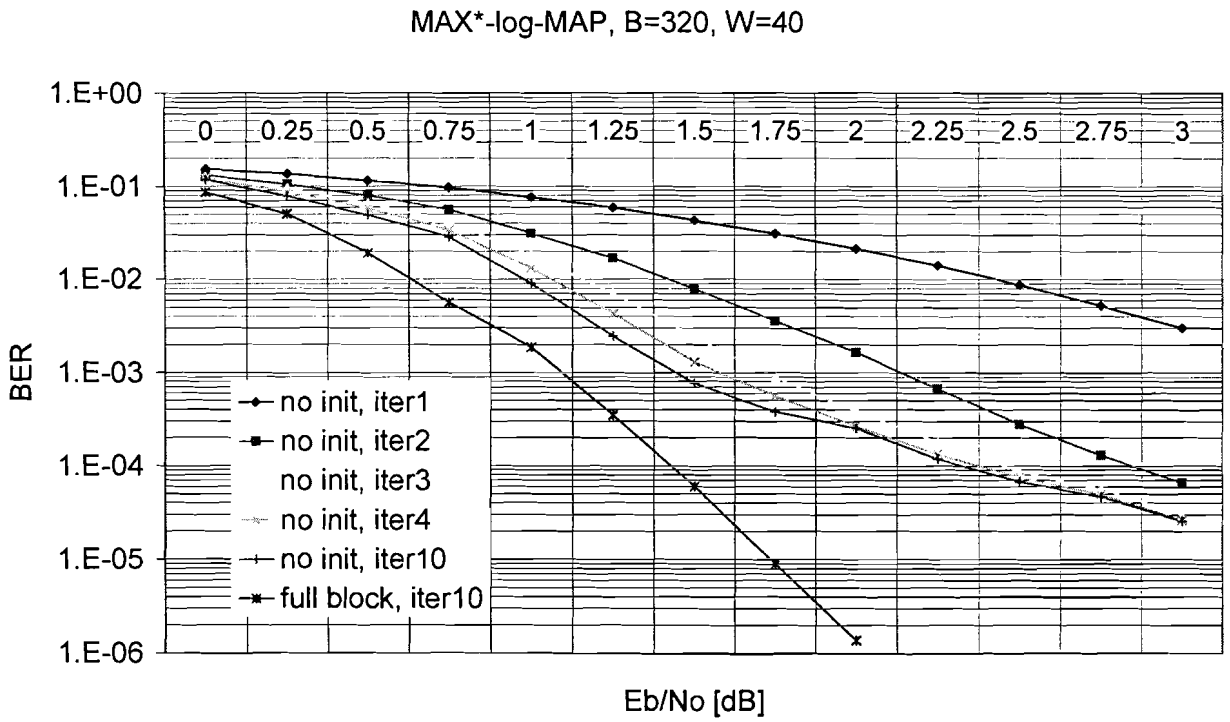

Figure A.3: BER versus Eb/No for different iterations, when using an all zero vector for initialisation

When using the Next Iteration Initialisation (NII) schedule (Figures A.4 and A.5), BER performance differences between the full block and the NII schedule are negligible after 10 iterations. Even after 6 iterations performance differences are less than 0.05dB. In the first iterations we see a difference in performance. The reason for this performance difference is the uniform state metric vector filled in at the stakes in the first iteration.

Figures A.6 and A.7 show the difference in performance in the first iterations. We can see that although the differences in performance between NII and full block are large (0.25dB), the performance of NII with half an iteration more is of the same order now in the advantage of NII. On average a SISO decoder has to do a quarter of an iteration more to achieve the same BER performance level. The implication of this result is that when the implemention cost [1] of a better initialisation in the first iteration are higher than the implementation cost of a quarter of an iteration, it is better to have no initialisation in the first iteration. The fact that the differences in performance narrow in each iteration confirms this statement, since at the end no performance differences remain.

The four figures A.4, A.5, A.6, and A.7 show that for larger block-lengths the initial difference between full block and NII are larger. Simulations show that for larger block-lengths NII performance converges sooner to full block performances.

If the window-length is increased from 40 to 64, an extra performance gain in the first iterations is observed. Figure A.8, A.10, A.9, and A.11 show the performance differences between window length 40 and 64 in the first iterations. The BER and WER gain of larger window-lengths for larger block-length is slightly higher.

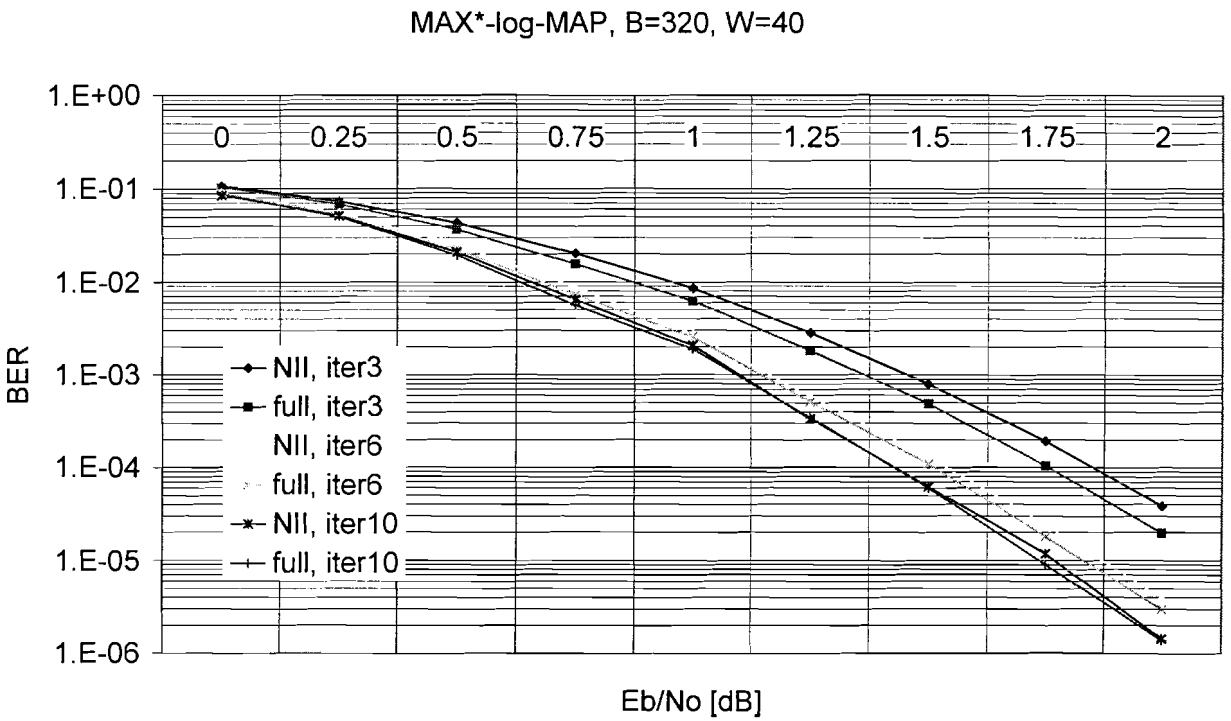

Figure A.4: BER versus Eb/No showing performance degradation of NII schedule

[1] implementation costs can be chip area, power and/or latency

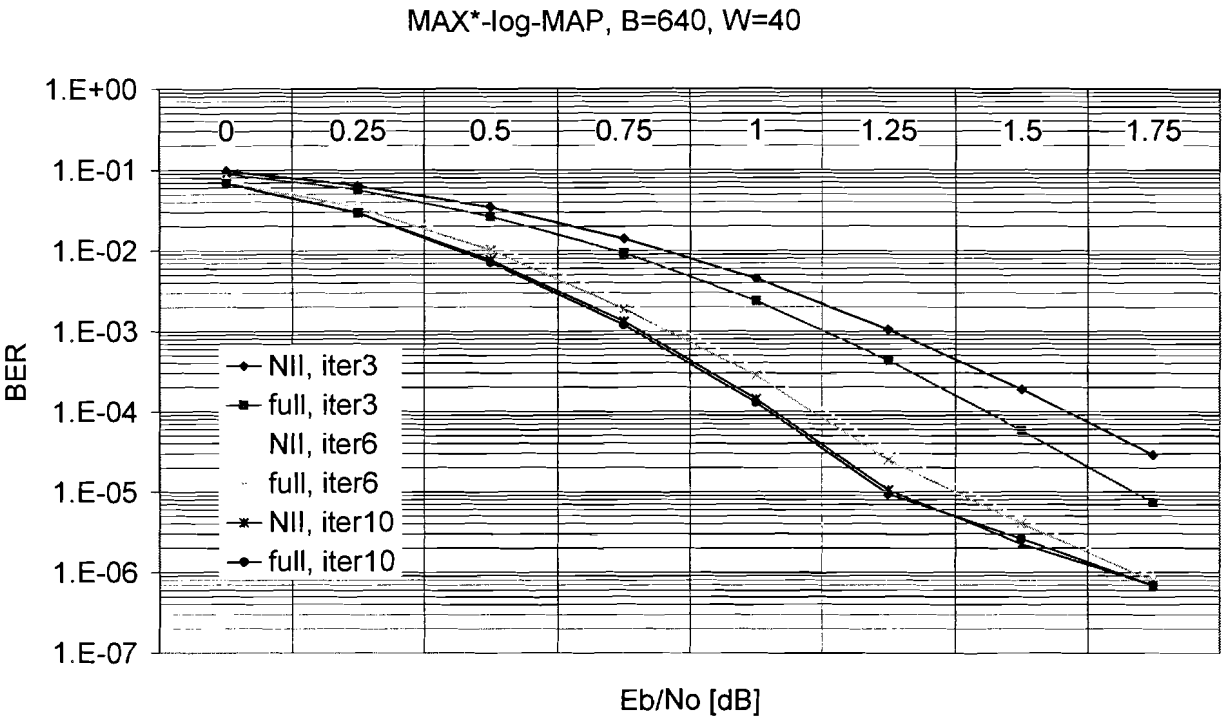

Figure A.5: BER versus Eb/No showing performance degradation of NII schedule

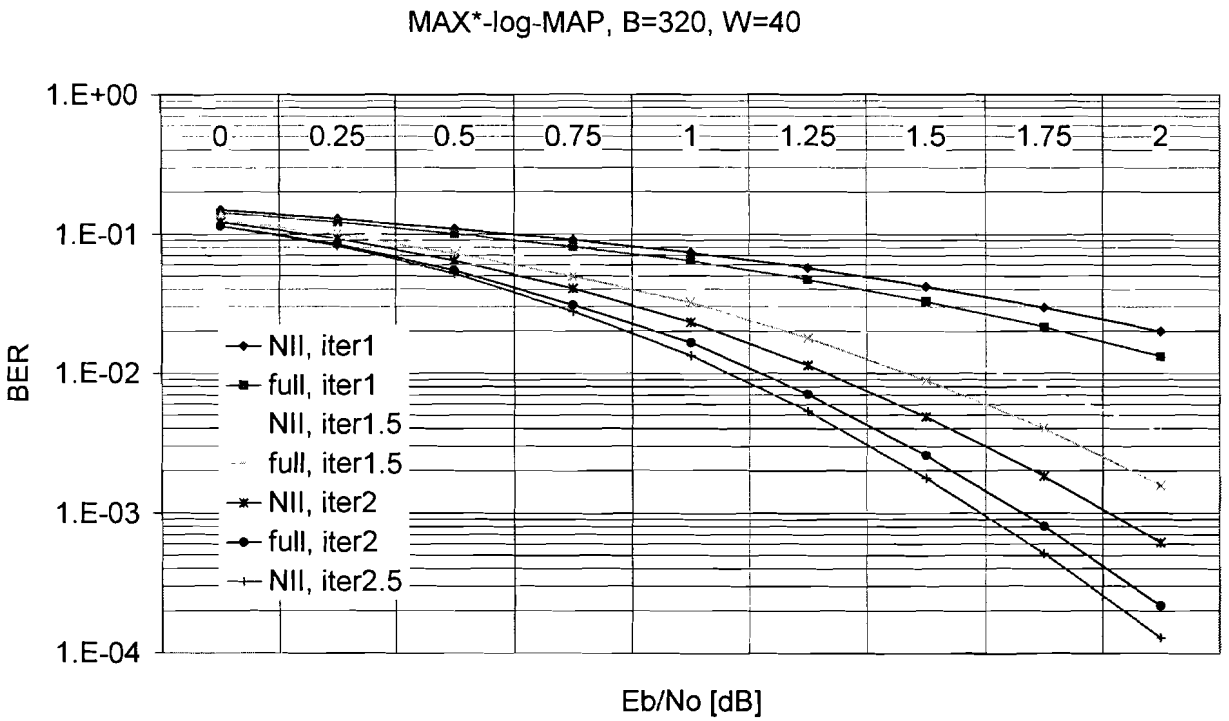

Figure A.6: BER versus Eb/No showing performance degradation in first iterations

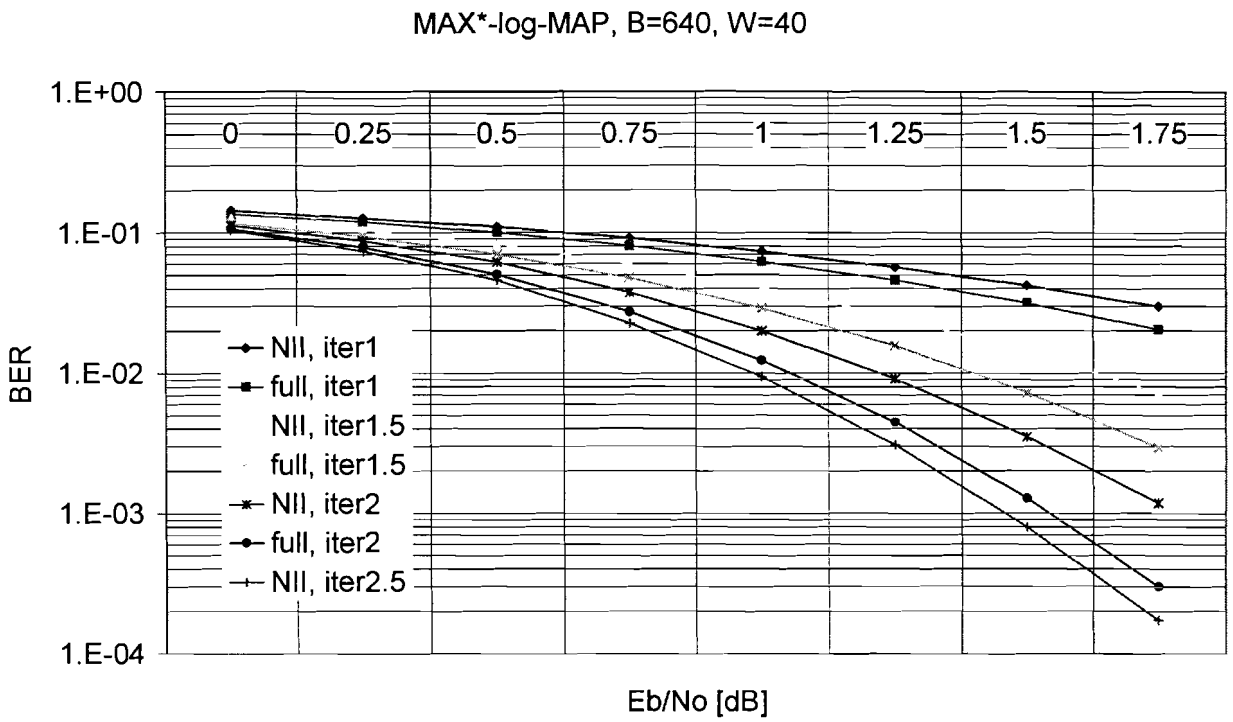

Figure A.7: BER versus Eb/No showing performance degradation in first iterations

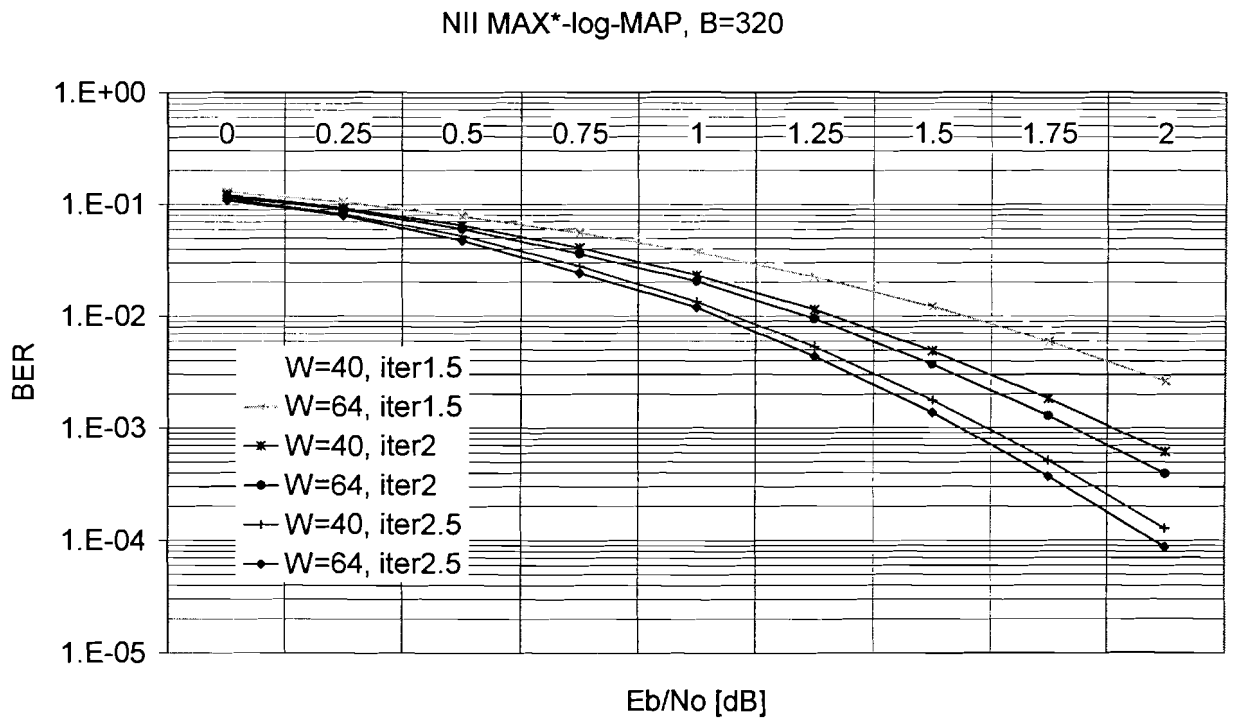

Figure A.8: BER versus Eb/No showing influence of window length in first iterations

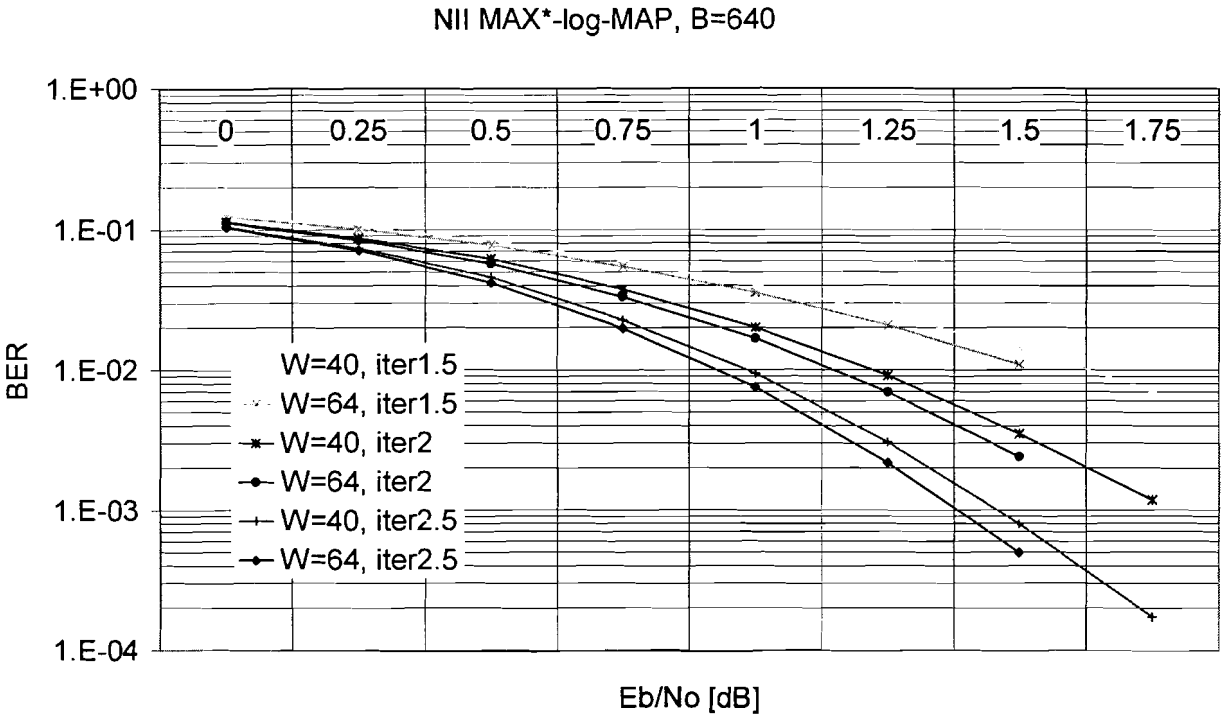

Figure A.9: BER versus Eb/No showing influence of window length in first iterations

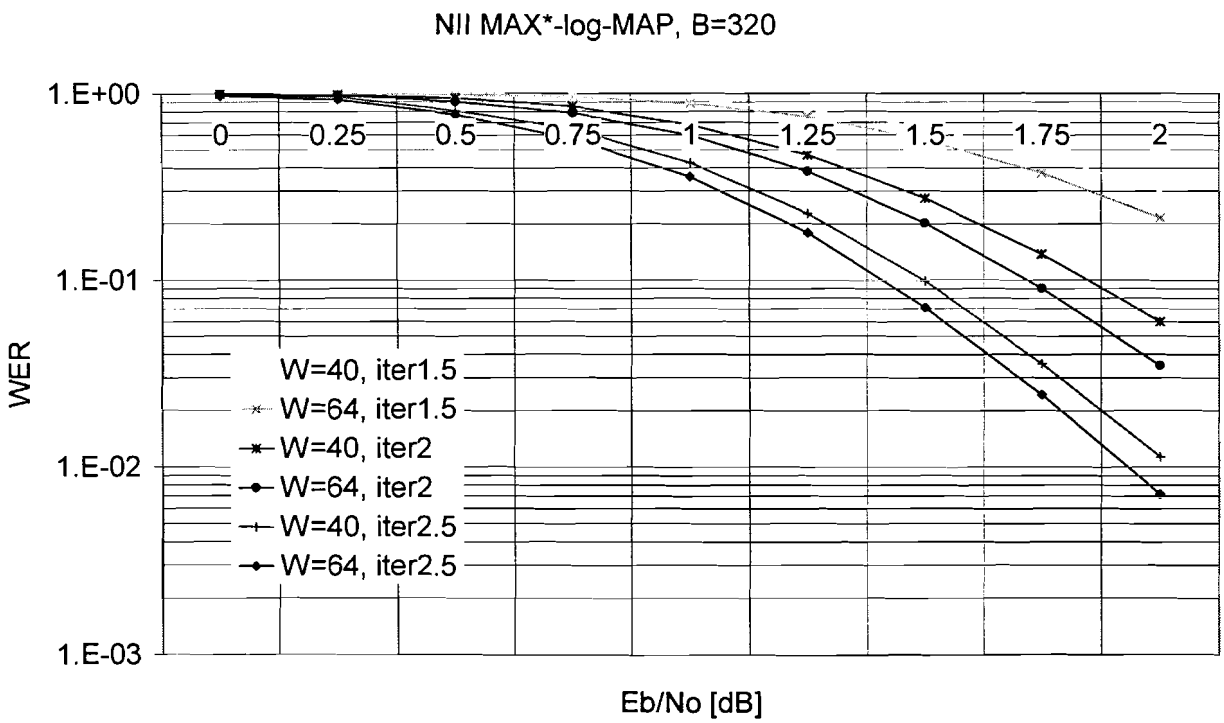

Figure A.10: WER versus Eb/No showing influence of window length in first iterations

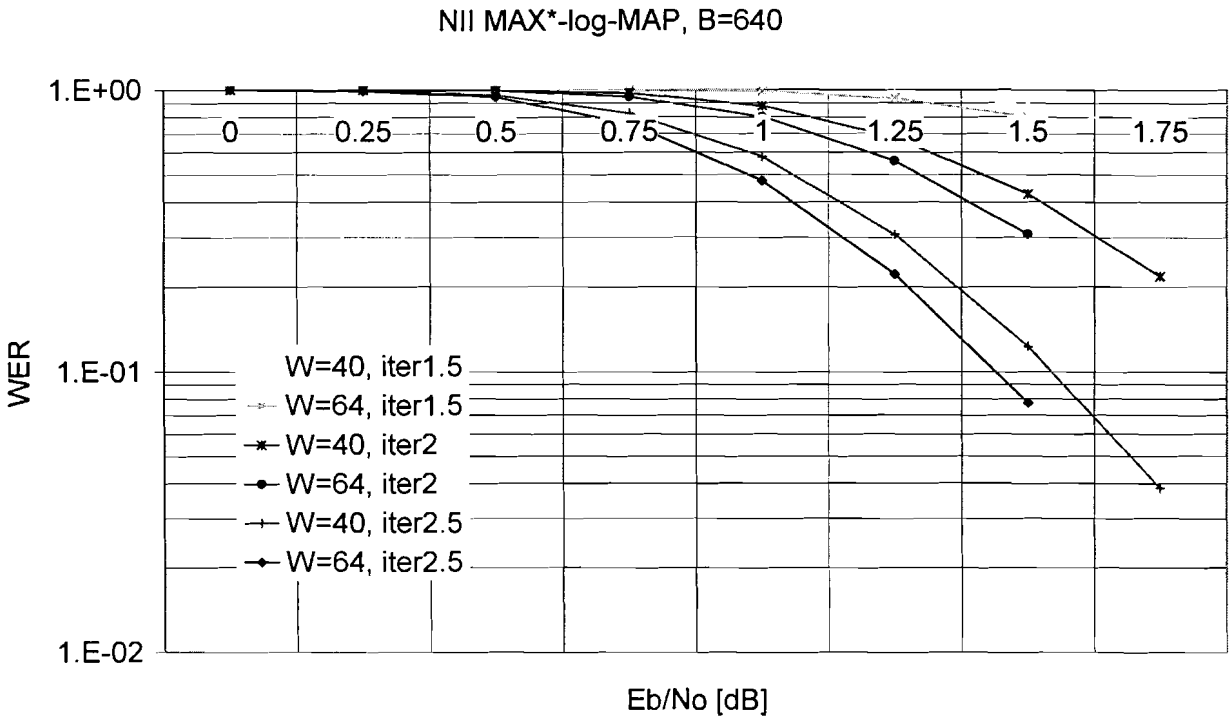

Figure A.11: WER versus Eb/No showing influence of window length in first iterations

Simulations point out that reversed order NII has negligible performance gain over NII. Therefore reversed order NII is not included in this report. The explanation for the lack of performance gain is that 40 training steps (from the previous iteration) are enough to cancel the additional error made two iterations ago.

Conclusions

- for larger iteration numbers NII has equal performance to full block scheme
- influence of window-length in the first iterations is measurable
- for larger block-lengths, performance differences in the first iteration is larger, but they convert earlier to full block performances
- first iteration initialisation has only advantages if the implementation costs are less than a quarter of a iteration cost.

Open points

- what are the effects of scaling the stored vector in the NII implemention technique on the BER performance?
- what are the Rayleigh fading channel performances of the NII implementation technique?
- Are there BER performance advantages using the reverse order NII implementation technique for smaller window lengths and/or Rayleigh fading AWGN channels?

A.4 Effects of optimizing window initialisation

In this section the effects of compressing the vector saved at the stakes are stated. Compressing the vector might be interesting, because each vector uses at least 56 bits (for 3GPP). The vector size is compressed to 3 bits, which represents the number of the state having the highest state metric. Simulation results have the name cNII, which stands for compressed Next Iteration Initialisation. Problems arise when recovering the vector for initialisation. At this point an error is made. Figure A.12 and A.13 show performance degradation when we reconstruct the vector, using the value 0 for the state with the highest state metric and X for all other states: (0, X). Within the range of our simulations it can be seen that in the 3^{rd} iteration the lower absolute value of X always has better performance, while in iteration 10 the performance lines cross. Further studies have to point out what values for X produce the highest performance and what criteria are best for scaling this factor X. Possible criteria are:

- iteration number
- estimation of Eb/No
- estimation of BER, already necessary for early stopping criteria.

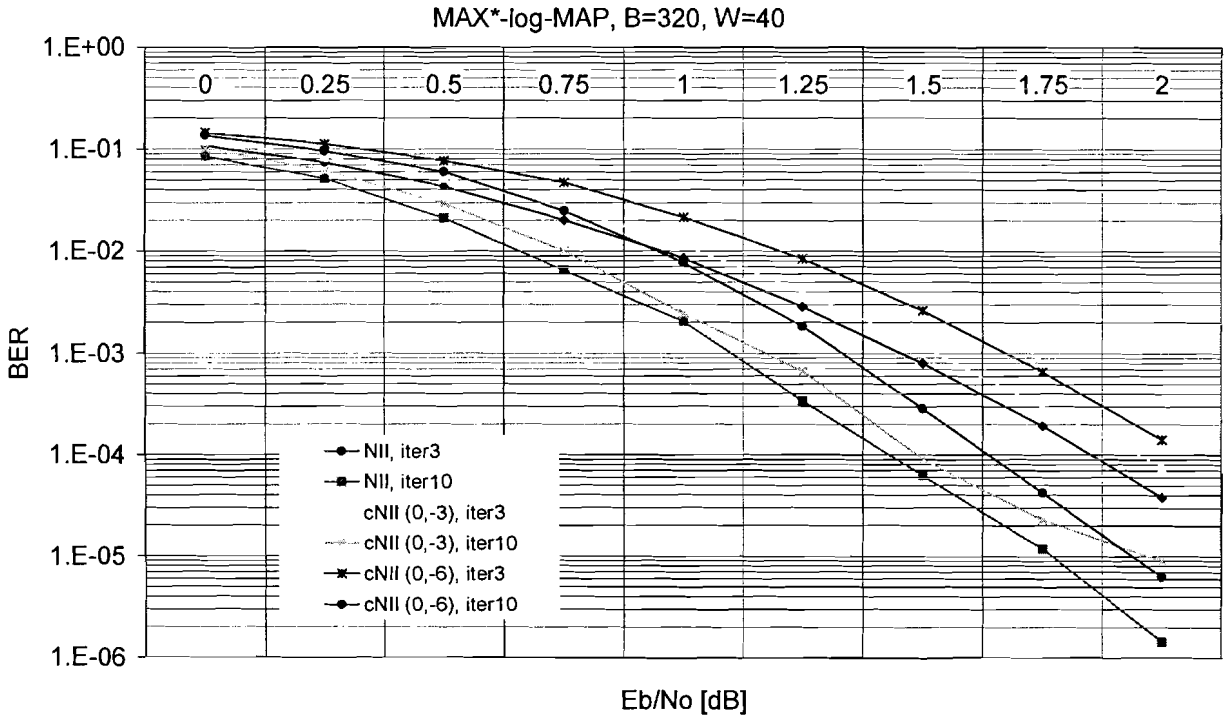

Figure A.12: BER versus Eb/No, showing performance degradation of vector compression, using a (0, X) vector for recovering

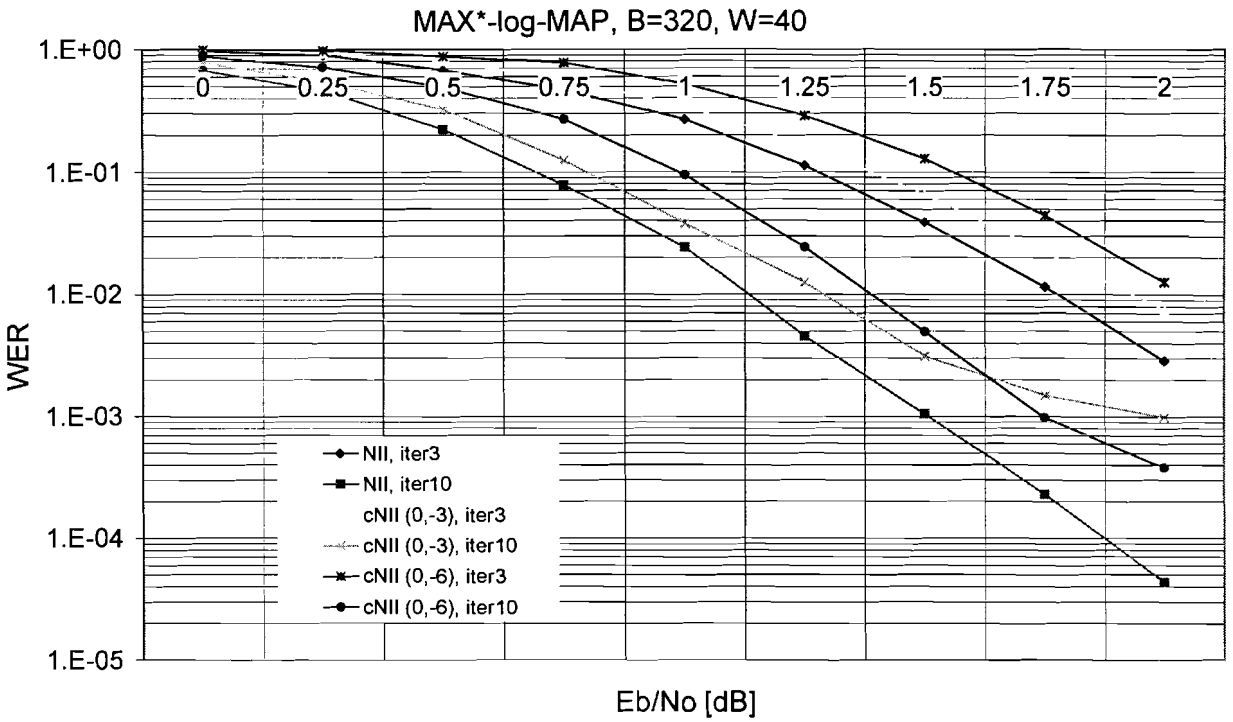

Figure A.13: WER versus Eb/No, showing performance degradation of vector compression, using a (0, X) vector for recovering

In Figures A.14 and A.15 we use a scalable eigen-vector for reconstruction of the vector. The eigen-vector is the state metric vector of trellis-steps, when no noise is added to the signal. There are three parameters for the eigen-vector:

- the number of the state having the highest state metric
- the magnitude of the systematic LLR
- the magnitude of the parity LLR

The first parameter is extracted from the previous iterations, the second and third parameter are choosen equal to X. This eigen-vector is further noted as e[X].
We can see that performance loss between NII and cNNI using the eigen-vector scheme, is less than the performance loss using the (0, X) scheme. Further studies have to find out:

- what values for X result in high performance
- what the gain is for not choosing the two magnitudes equal
- which scaling criteria result in high performance. Alternatives are:
  - iteration number
  - Eb/No
  - magnitude of LLRs

Simulations in this report show that using the first criteria is a good idea. In the first iterations, a lower scaling factor gives better performance. For higher iteration numbers, BERs can be found for which higher scaling factors reduce this BER.
These studies also have to point out what the gap between the (0, X) and e[X] is.

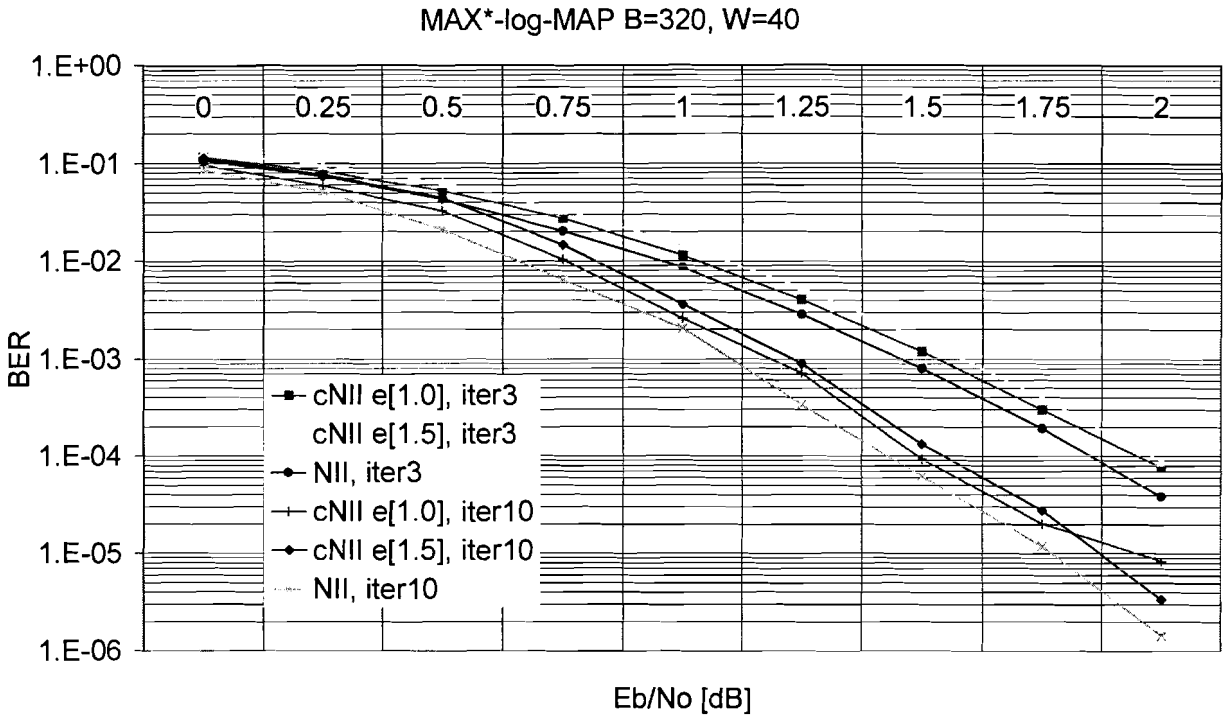

Figure A.14: BER versus Eb/No, showing performance degradation of vector compression, using the eigenvector for recovering

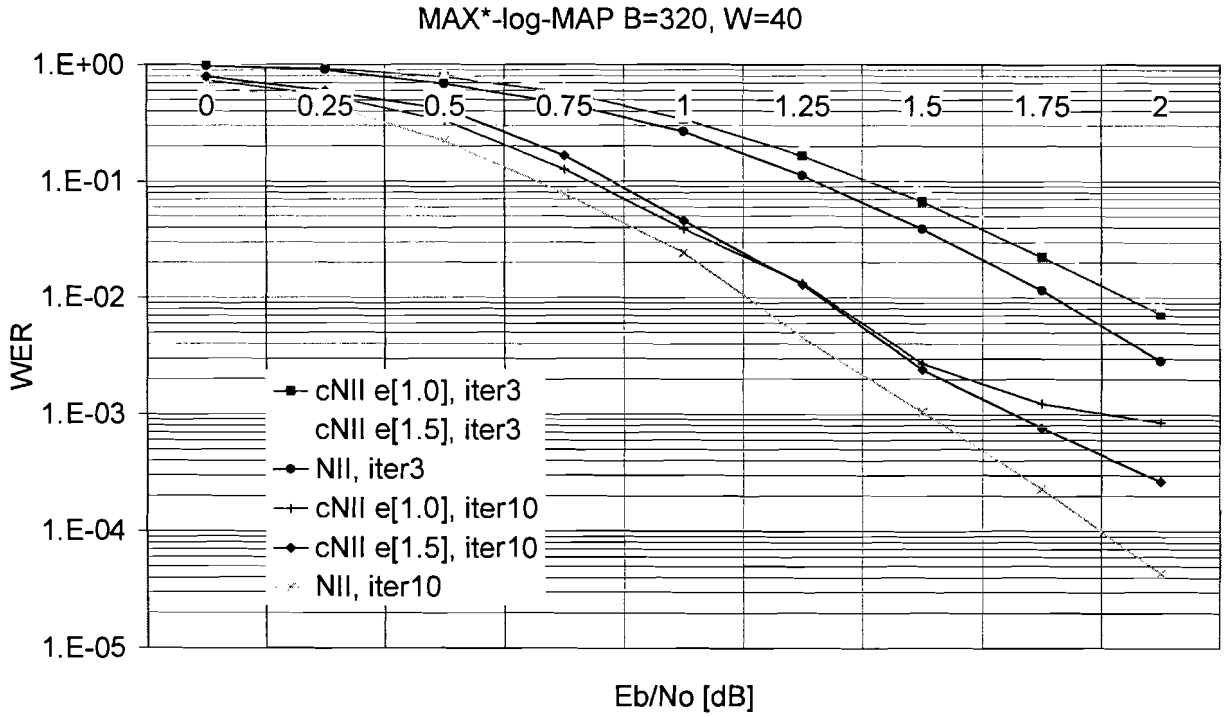

Figure A.15: WER versus Eb/No, showing performance degradation of vector compression, using the eigenvector for recovering

Figures A.16 and A.17 show the influence of a combined training calculation cNII scheme. Vectors at stakes B+t, B-W+t, $\cdots$, W+t are stored. After reconstruction of the vector t training calculations are done. Note that for $t > \mu$ an exception for stake B+t has to be made. Although performance gain is observed, it remains an open question if this combined scheme is worth the extra implementation cost.

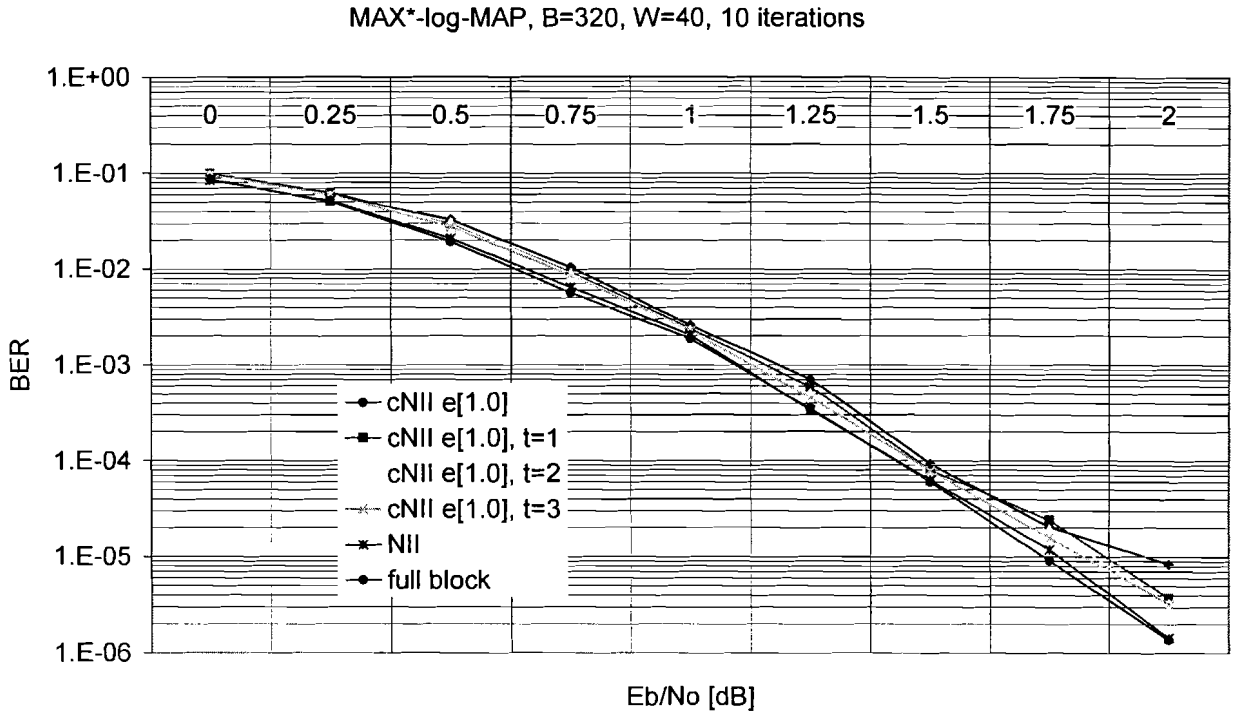

Figure A.16: BER versus Eb/No, showing performance influence of training steps, combined with compressed NII

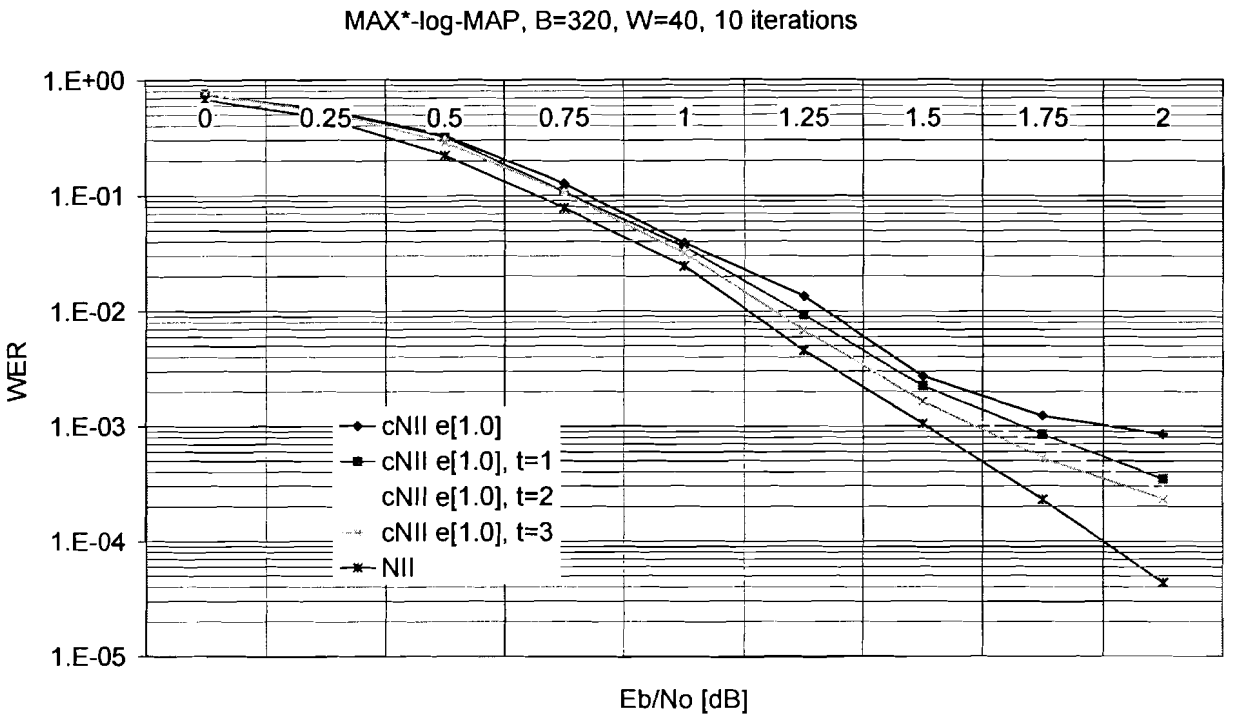

Figure A.17: WER versus Eb/No, showing performance influences of training steps, combined with compressed NII

Figures A.18 and A.19 show the results of using eigen-vector reconstruction for block-length 640 bits. For low Eb/No it is better to use a lower scaling factor. Several crosslines can be seen in the 10^{th} iteration curves. In the 3^{rd} iterations the lowest scaling factor always has the best performance.

Figure A.20 shows the results for a scalable eigen-vector. It compares the eigen-vector performances for two different block-lengths. Performance of the eigen-vector scheme seems to work out better for larger block-lengths. Care has to be taken at this conclusion, because not all scaling schemes and scaling values have been tried out. When comparing the WER of the eigen-vector scheme (Figure A.15 and A.19), we can see that for larger block-lenghts the WER performance decreases. More simulations have to be done to confirm this observation.

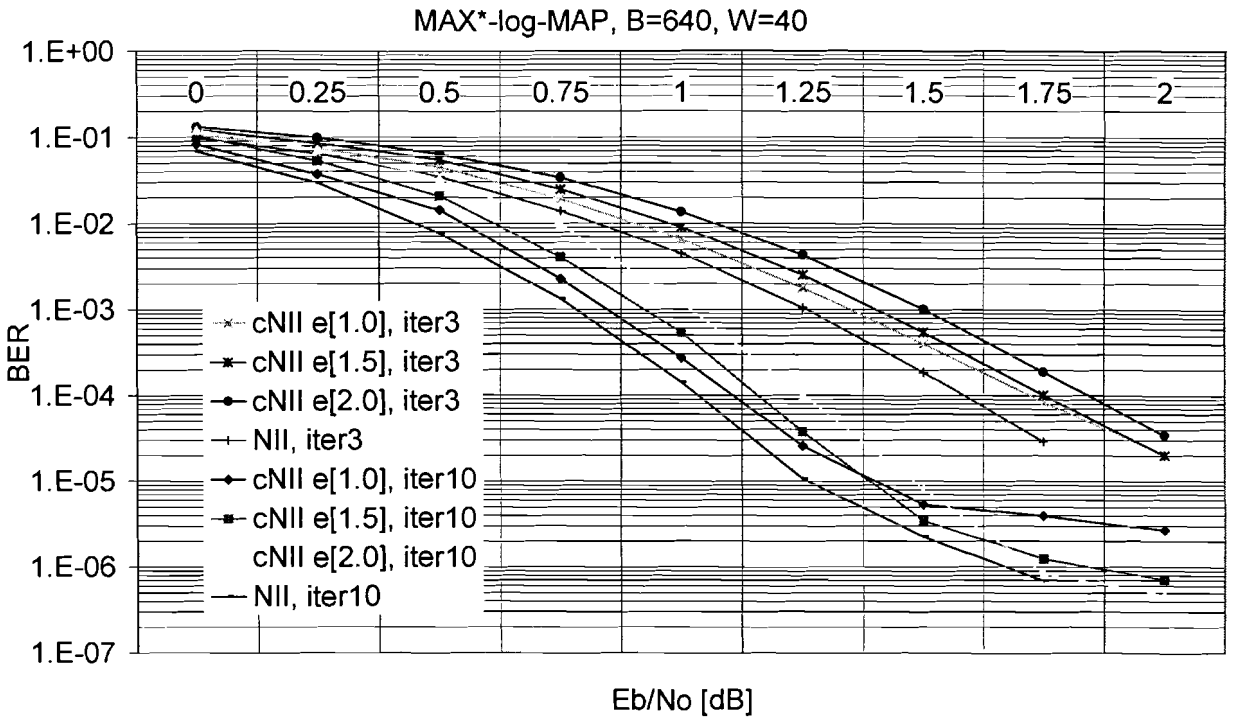

Figure A.18: BER versus Eb/No, showing performance degradation of vector compression, using a eigen-vector for recovering

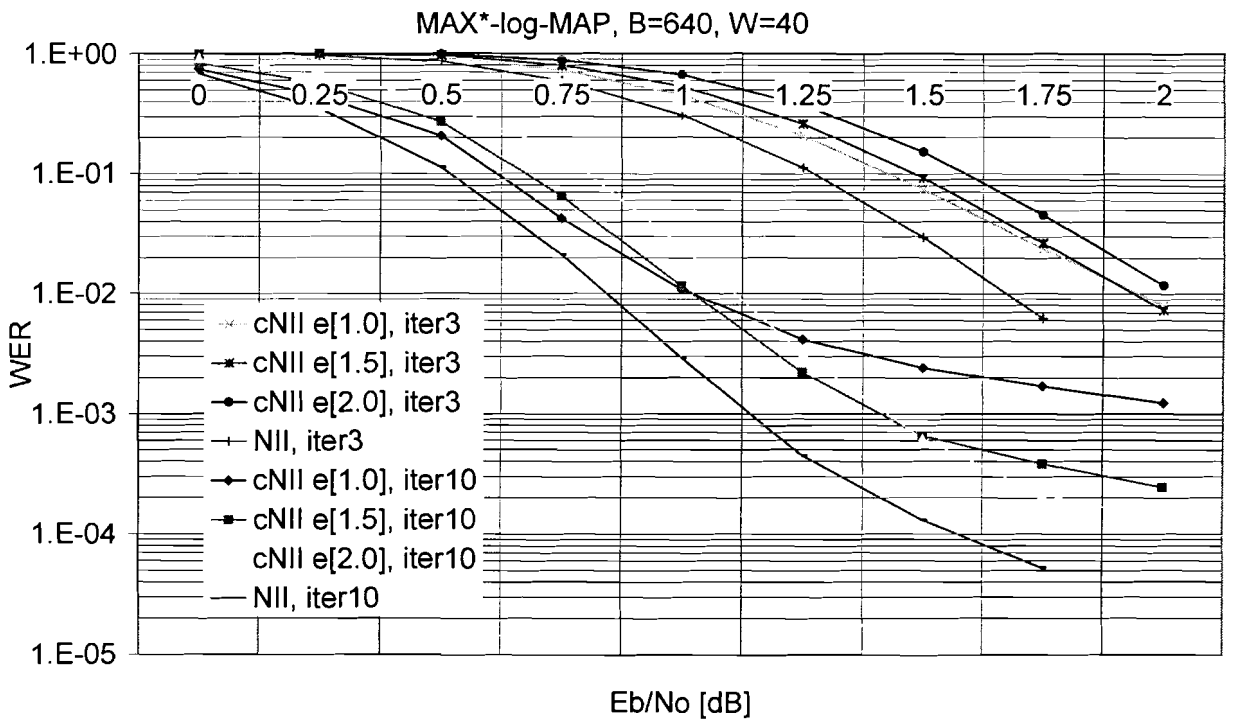

Figure A.19: WER versus Eb/No, showing performance degradation of vector compression, using a eigen-vector for recovering

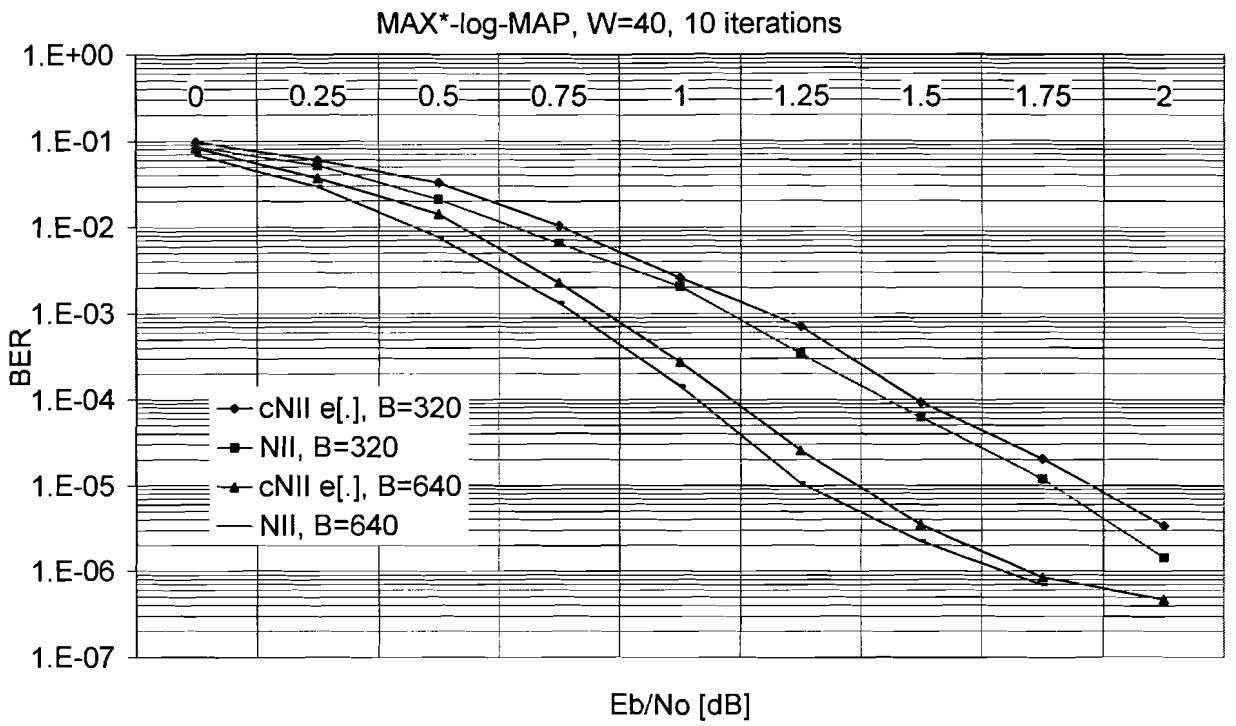

Figure A.20: BER versus Eb/No, comparing the effects of block-length on performance differences of vector compression

Conclusions

- BER performance loss due to vector compression is less than 0.1 dB
- larger block-lengths have positive effects on BER performance and negative effects on WER of vector compression

Open points

- what is the best scaling criterion for reconstructing the state metric?
- what scaling values result in the best BER performance?
- what is the gain of using 2 different magnitudes as parameter for the eigen-vector?
- what is the relation between window length W and BER/WER performance for cNII?
- for variable block-length implementations like 3GPP it might be interesting to use a combined NII, cNII scheme. What is the influence of such a scheme on the BER performances?
- what are the Rayleigh fading AWGN channel performances of the cNII implementation technique?

A.5 Effects of scaling extrinsic information

One of the algorithm variants on turbo decoding level is scaling the extrinsic information. Since the gain for the Max-log-MAP algorithm is larger than for the Max⋆-log-MAP algorithm, the influence of scaling Max-log-MAP is shown. The scaling factor for the scaled Max-log-MAP variant is set to 0.625. Varying this scaling factor will give additional performance improvement. Several criteria for varying the scaling factor can be applied, including:

- iteration number
- Eb/No estimation
- BER estimation
- magnitude of extrinsic systematic information (non-linear scaling)
- block-length

Figure A.21 shows the differences in performance for the 3 algorithms after 3 and 10 iterations. The performance gap between Max⋆ - and Max- log-MAP is 0.4 dB. The performance gap between Max⋆ -log-MAP and scaled Max-log-MAP is 0.125 dB (BER=10^{-3}). For higher iteration numbers and lower BER, the performance gap reduces and eventually disappears. Note that the WER performance gap (figure A.22) remains constant at approximately 0.07 dB.

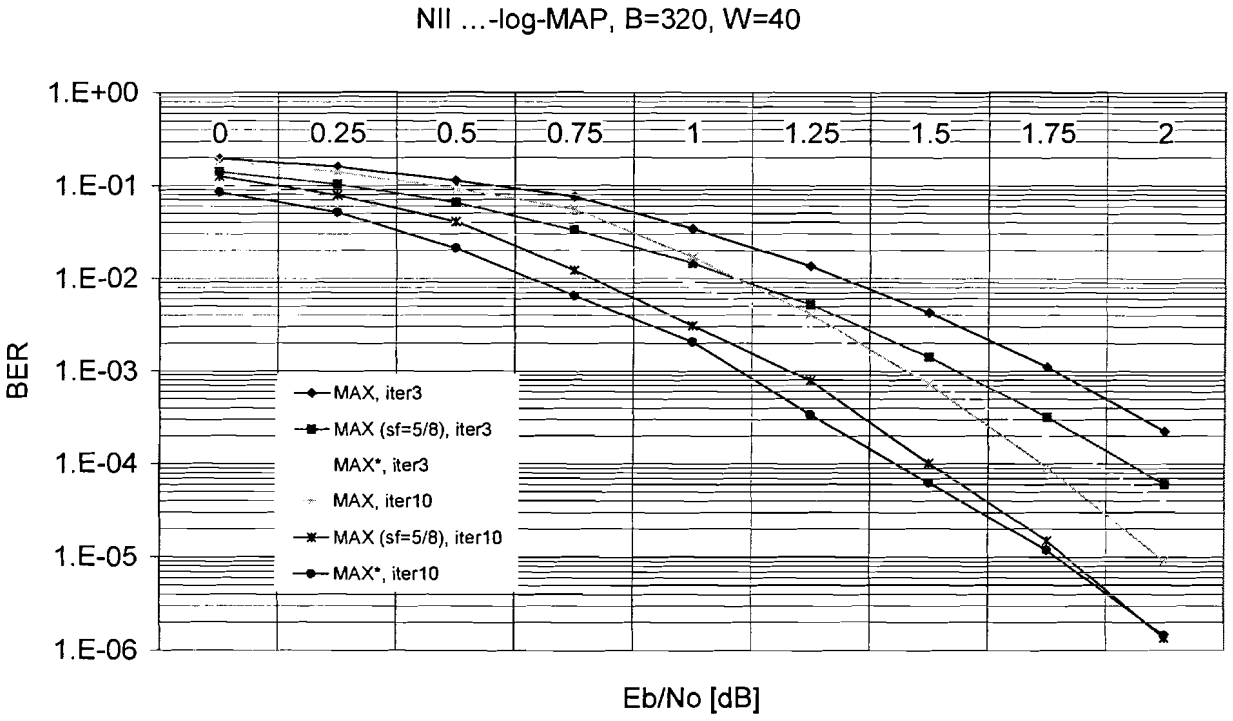

Figure A.21: BER versus Eb/No, showing performance difference between Max⋆ -, Max - with scaling and Max-log-MAP algorithms

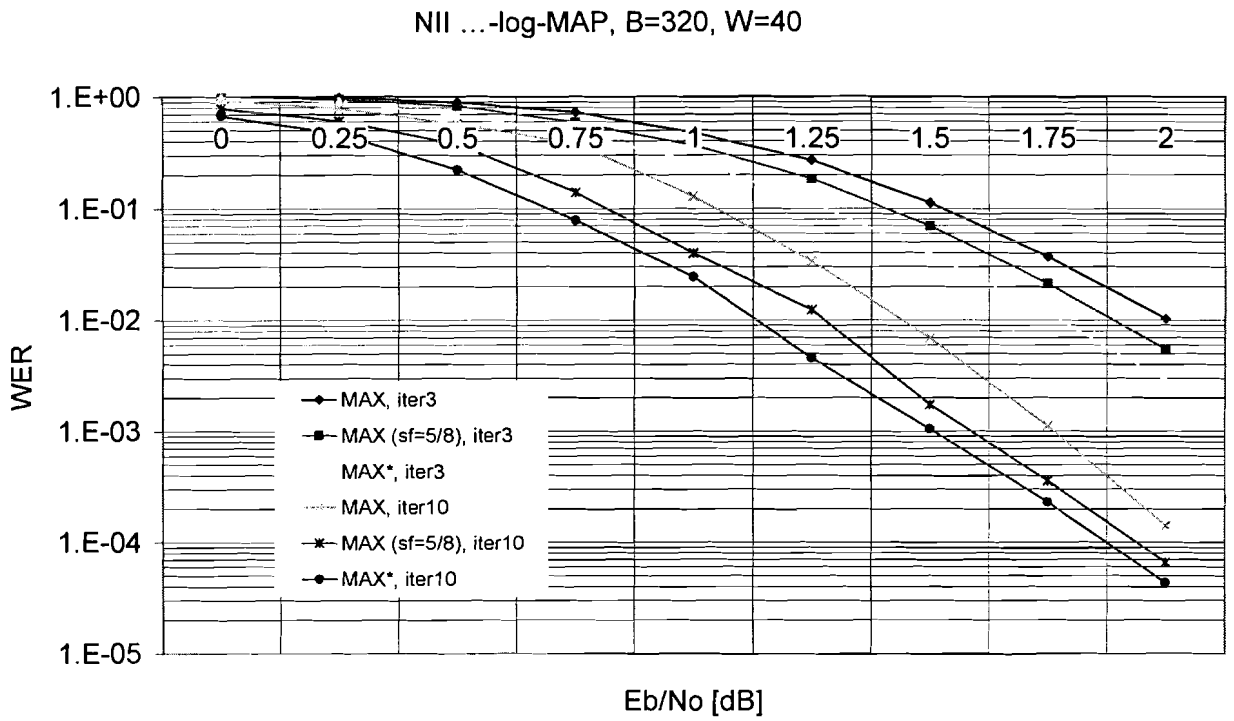

Figure A.22: WER versus Eb/No, showing performance difference between Max$\star$ -, Max - with scaling and Max-log-MAP algorithms

Figure A.23 shows the influence of scaling on a block-length of 640 bits. The same conclusions as for block-length 320 bits can be applied. An interesting detail is that scaled Max-log-MAP outperforms Max$\star$-log-MAP at BER of 10^{-6}. The WER of scaled Max-log-MAP remains higher than Max$\star$-log-MAP in our simulations. Although it is theoretically not possible to outperform BCJR (Max$\star$-log-MAP) for a convolution code, within the Turbo decoding scheme it is possible. An explanation might be found in the correlation between intrinsic systematic information and extrinsic systematic information. The output of SISO 2 might contain too much direct information from RSC 1 and feed-back would result in performance degradation. A scaling factor reduces this correlation.

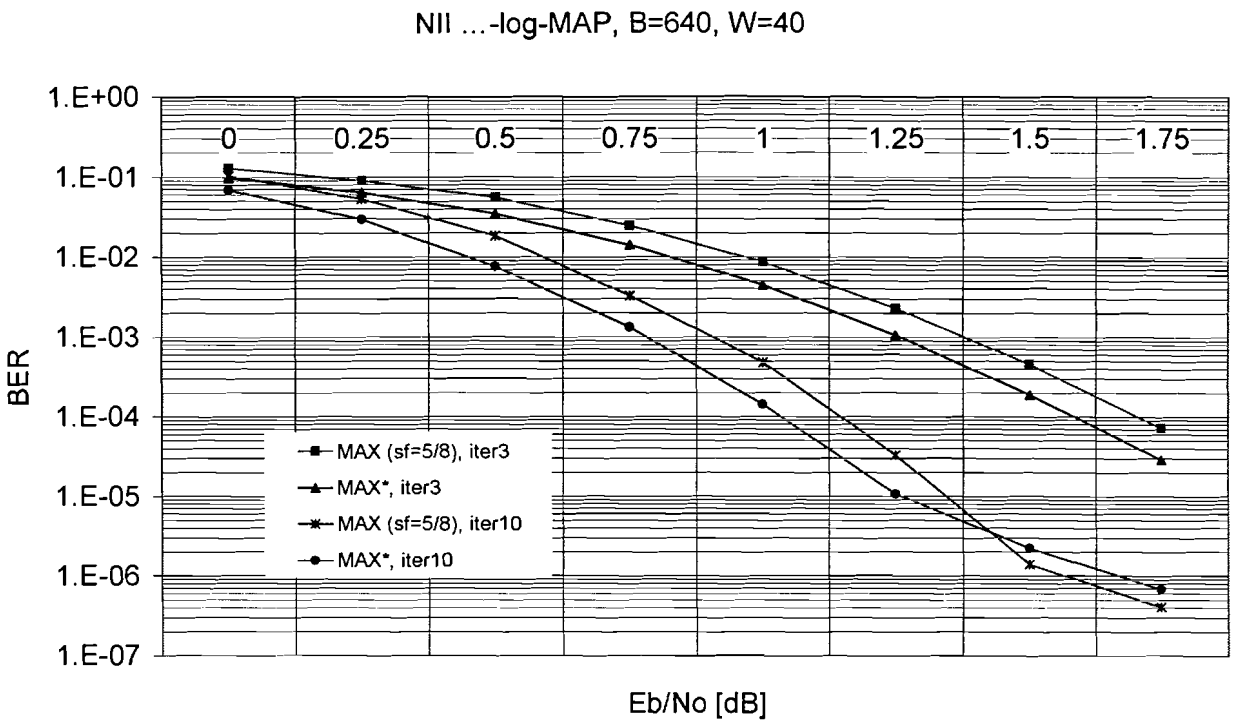

Figure A.23: BER versus Eb/No, showing performance difference between Max⋆ -, Max - with scaling and Max-log-MAP algorithms

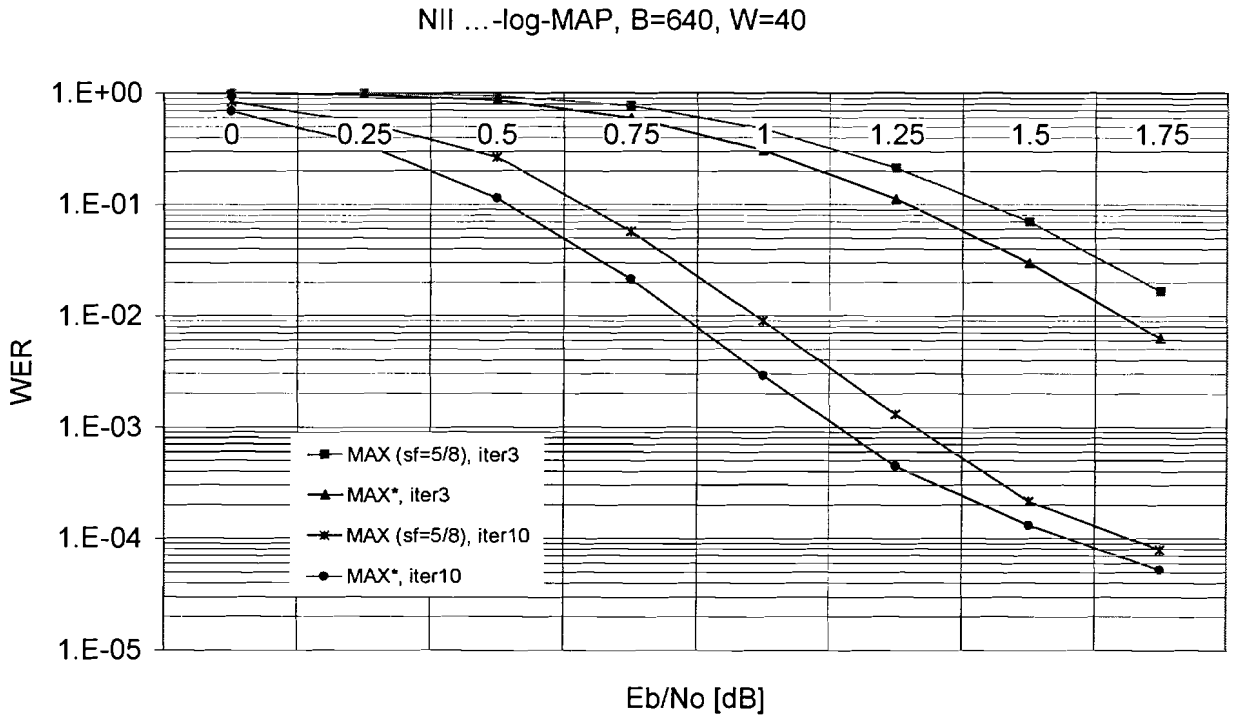

Figure A.24: WER versus Eb/No, showing performance difference between Max⋆ -, Max - with scaling and Max-log-MAP algorithms

Conclusions

- scaled Max-log-MAP has negligible performance loss for low BER

Open points

- what are the best criteria to vary scaling factor *sf*?
- which scaling factors *sf* achieve the best BER/WER performance
- what are the effects of scaling extrinsic information of the Max$\star$-log-MAP?

Appendix B

Eigen-vector

This chapter explains the eigen vector calculations.
The observation during simulations of the trellis was: if for example the all zero code word (all negative values) is used, then the state metric vector becomes a constant vector. This vector is only scaled by the magnitude of the input samples. When traveling through the trellis, and normalising either to state zero always being zero or the highest state metric always being zero the following observations have been made:

- the difference between state metrics was maximal when traveling on the ideal path, with a maximum amplitude
- each trellis step has its own eigen-vector, when travelling on the ideal path, with a constant input amplitude
- when the input amplitude is constant the eigen-vector only depends on which state has the highest state metric
- each eigen-vector is composed of the same state metric values
- forward recursion α and backward recursion β lead to other eigen-vectors

Eigen-vectors can be used for the following aims:

- determing the maximum difference between state metrics
- avoiding larger difference due to incorrect initialisation (for example: (0, -M, ... ,-M), which can lead to differences larger than M)
- initialising sliding window recursions in the cNII implementation technique. When in the previous iteration state k has the highest state metric at a stake, there are states which are more likely (have higher state metics) than others. When using an eigen-vector for vector reconstruction, these differences can be exploid.

For calculating the backward recursion eigen-vector belonging to state zero having the highest state metric the next assumptions are made:
$\lfloor \lambda i^s \rfloor < 0$
$\lfloor \lambda i^p \rfloor < 0$
$\lfloor \lambda i^s \rfloor = \lfloor \lambda i^p \rfloor$: the intrinsic LLR come from the same source and it is therefore justifible to choose the 'number space' equal.
$\lfloor \lambda a^s \rfloor \leq 0$

β_0 has the highest state metric
$\beta_0 = 0$

. $\otimes$. summation of left and right operand
. $\oplus$. maximum of left and right operand
Both operations are commutative. $\otimes$ operations have a higher priority than the $\oplus$ operations

and:

$$\beta_0(i) = \beta_0(i+1) \oplus \beta_4(i+1) \otimes \lfloor \lambda i^s \rfloor \otimes \lfloor \lambda a^s \rfloor \otimes \lfloor \lambda i^p \rfloor = \beta_0(i+1) \quad \text{(B.1)}$$
$$\beta_1(i) = \beta_0(i+1) \otimes \lfloor \lambda i^s \rfloor \otimes \lfloor \lambda a^s \rfloor \otimes \lfloor \lambda i^p \rfloor \oplus \beta_4(i+1) = \beta_1(i+1) \quad \text{(B.2)}$$
$$\beta_2(i) = \beta_1(i+1) \otimes \lfloor \lambda i^s \rfloor \otimes \lfloor \lambda a^s \rfloor \oplus \beta_5(i+1) \otimes \lfloor \lambda i^p \rfloor = \beta_2(i+1) \quad \text{(B.3)}$$
$$\beta_3(i) = \beta_1(i+1) \otimes \lfloor \lambda i^p \rfloor \oplus \beta_5(i+1) \otimes \lfloor \lambda i^s \rfloor \otimes \lfloor \lambda a^s \rfloor = \beta_3(i+1) \quad \text{(B.4)}$$
$$\beta_4(i) = \beta_2(i+1) \otimes \lfloor \lambda i^p \rfloor \oplus \beta_6(i+1) \otimes \lfloor \lambda i^s \rfloor \otimes \lfloor \lambda a^s \rfloor = \beta_4(i+1) \quad \text{(B.5)}$$
$$\beta_5(i) = \beta_2(i+1) \otimes \lfloor \lambda i^s \rfloor \otimes \lfloor \lambda a^s \rfloor \oplus \beta_6(i+1) \otimes \lfloor \lambda i^p \rfloor = \beta_5(i+1) \quad \text{(B.6)}$$
$$\beta_6(i) = \beta_3(i+1) \otimes \lfloor \lambda i^s \rfloor \otimes \lfloor \lambda a^s \rfloor \otimes \lfloor \lambda i^p \rfloor \oplus \beta_7(i+1) = \beta_6(i+1) \quad \text{(B.7)}$$
$$\beta_7(i) = \beta_3(i+1) \oplus \beta_7(i+1) \otimes \lfloor \lambda i^s \rfloor \otimes \lfloor \lambda i^s \rfloor \otimes \lfloor \lambda i^p \rfloor = \beta_7(i+1) \quad \text{(B.8)}$$

These assumptions lead to the next 8 equations:

$$\beta_0 = 0 \quad \text{(B.9)}$$
$$\beta_1 = \lfloor \lambda i^s \rfloor \otimes \lfloor \lambda a^s \rfloor \otimes \lfloor \lambda i^p \rfloor \oplus \beta_4 \quad \text{(B.10)}$$
$$\beta_2 = \beta_1 \otimes \lfloor \lambda i^s \rfloor \otimes \lfloor \lambda a^s \rfloor \oplus \beta_5 \otimes \lfloor \lambda i^p \rfloor \quad \text{(B.11)}$$
$$\beta_3 = \beta_1 \otimes \lfloor \lambda i^p \rfloor \oplus \beta_5 \otimes \lfloor \lambda i^s \rfloor \otimes \lfloor \lambda a^s \rfloor \quad \text{(B.12)}$$
$$\beta_4 = \beta_2 \otimes \lfloor \lambda i^p \rfloor \oplus \beta_6 \otimes \lfloor \lambda i^s \rfloor \otimes \lfloor \lambda a^s \rfloor \quad \text{(B.13)}$$
$$\beta_5 = \beta_2 \otimes \lfloor \lambda i^s \rfloor \otimes \lfloor \lambda a^s \rfloor \oplus \beta_6 \otimes \lfloor \lambda i^p \rfloor \quad \text{(B.14)}$$
$$\beta_6 = \beta_3 \otimes \lfloor \lambda i^s \rfloor \otimes \lfloor \lambda a^s \rfloor \otimes \lfloor \lambda i^p \rfloor \oplus \beta_7 \quad \text{(B.15)}$$
$$\beta_7 = \beta_3 \oplus \beta_7 \otimes \lfloor \lambda i^s \rfloor \otimes \lfloor \lambda a^s \rfloor \otimes \lfloor \lambda i^p \rfloor = \beta_3 \quad \text{(B.16)}$$

Note that in the last equation the next property is used:
$\beta_7 > \beta_7 \otimes \lfloor \lambda i^s \rfloor \otimes \lfloor \lambda a^s \rfloor \otimes \lfloor \lambda i^p \rfloor$. This property is applied often in the next equations and will be addressed as: removing terms containing β_n.

$$\beta_1 = \lfloor \lambda i^s \rfloor \otimes \lfloor \lambda a^s \rfloor \otimes \lfloor \lambda i^p \rfloor \oplus \beta_4 \quad \text{(B.17)}$$
$$\beta_2 = \beta_1 \otimes \lfloor \lambda i^s \rfloor \otimes \lfloor \lambda a^s \rfloor \oplus \beta_5 \otimes \lfloor \lambda i^p \rfloor \quad \text{(B.18)}$$
$$\beta_3 = \beta_1 \otimes \lfloor \lambda i^p \rfloor \oplus \beta_5 \otimes \lfloor \lambda i^s \rfloor \otimes \lfloor \lambda a^s \rfloor \quad \text{(B.19)}$$
$$\beta_4 = \beta_2 \otimes \lfloor \lambda i^p \rfloor \oplus \beta_6 \otimes \lfloor \lambda i^s \rfloor \otimes \lfloor \lambda a^s \rfloor \quad \text{(B.20)}$$
$$\beta_5 = \beta_2 \otimes \lfloor \lambda i^s \rfloor \otimes \lfloor \lambda a^s \rfloor \oplus \beta_6 \otimes \lfloor \lambda i^p \rfloor \quad \text{(B.21)}$$
$$\beta_6 = \beta_3 = \beta_7 \quad \text{(B.22)}$$

In equation B.22 we use the next properties: $\beta_3 = \beta_7$ and $\beta_3 > \beta_3 \otimes \lfloor \lambda i^s \rfloor \otimes \lfloor \lambda a^s \rfloor \otimes \lfloor \lambda i^p \rfloor$.

In the next equations β_1 and β_5 are substituted with equation B.17 and B.21 respectively. β_3 is substituted for β_6.

$$\begin{aligned}
\beta_2 &= 2 \cdot (\lfloor \lambda i^s \rfloor \otimes \lfloor \lambda a^s \rfloor) \otimes \lfloor \lambda i^p \rfloor \oplus \beta_4 \otimes \lfloor \lambda i^s \rfloor \otimes \lfloor \lambda a^s \rfloor \oplus \\
&\quad \beta_2 \otimes \lfloor \lambda i^s \rfloor \otimes \lfloor \lambda a^s \rfloor \otimes \lfloor \lambda i^p \rfloor \oplus \beta_3 \otimes 2 \cdot \lfloor \lambda i^p \rfloor && \text{(B.23)} \\
\beta_3 &= \lfloor \lambda i^s \rfloor \otimes \lfloor \lambda a^s \rfloor \otimes 2 \cdot \lfloor \lambda i^p \rfloor \oplus \beta_4 \otimes \lfloor \lambda i^p \rfloor \oplus \\
&\quad \beta_2 \otimes 2 \cdot (\lfloor \lambda i^s \rfloor \otimes \lfloor \lambda a^s \rfloor) \oplus \beta_3 \otimes \lfloor \lambda i^s \rfloor \otimes \lfloor \lambda a^s \rfloor \otimes \lfloor \lambda i^p \rfloor && \text{(B.24)} \\
\beta_4 &= \beta_2 \otimes \lfloor \lambda i^p \rfloor \oplus \beta_3 \otimes \lfloor \lambda i^s \rfloor \otimes \lfloor \lambda a^s \rfloor && \text{(B.25)}
\end{aligned}$$

These equations can be simplified, removing the terms containing β_2 and β_3 in equations B.23 and B.24 respectively.

$$\begin{aligned}
\beta_2 &= 2 \cdot (\lfloor \lambda i^s \rfloor \otimes \lfloor \lambda a^s \rfloor) \otimes \lfloor \lambda i^p \rfloor \oplus \beta_4 \otimes \lfloor \lambda i^s \rfloor \otimes \lfloor \lambda a^s \rfloor \oplus \\
&\quad \beta_3 \otimes 2 \cdot \lfloor \lambda i^p \rfloor && \text{(B.26)} \\
\beta_3 &= \lfloor \lambda i^s \rfloor \otimes \lfloor \lambda a^s \rfloor \otimes 2 \cdot \lfloor \lambda i^p \rfloor \oplus \beta_4 \otimes \lfloor \lambda i^p \rfloor \oplus \\
&\quad \beta_2 \otimes 2 \cdot (\lfloor \lambda i^s \rfloor \otimes \lfloor \lambda a^s \rfloor) && \text{(B.27)} \\
\beta_4 &= \beta_2 \otimes \lfloor \lambda i^p \rfloor \oplus \beta_3 \otimes \lfloor \lambda i^s \rfloor \otimes \lfloor \lambda a^s \rfloor && \text{(B.28)}
\end{aligned}$$

The next step in the solution is to substitute β_4 of equation B.28 in the equations B.26 and B.27.

$$\begin{aligned}
\beta_2 &= 2 \cdot (\lfloor \lambda i^s \rfloor \otimes \lfloor \lambda a^s \rfloor) \otimes \lfloor \lambda i^p \rfloor \oplus \beta_2 \otimes \lfloor \lambda i^s \rfloor \otimes \lfloor \lambda a^s \rfloor \otimes \lfloor \lambda i^p \rfloor \oplus \\
&\quad \beta_3 \otimes 2 \cdot (\lfloor \lambda i^s \rfloor \otimes \lfloor \lambda a^s \rfloor) \oplus \beta_3 \otimes 2 \cdot \lfloor \lambda i^p \rfloor && \text{(B.29)} \\
&= 2 \cdot (\lfloor \lambda i^s \rfloor \otimes \lfloor \lambda a^s \rfloor) \otimes \lfloor \lambda i^p \rfloor \oplus \beta_3 \otimes 2 \cdot \lfloor \lambda i^p \rfloor && \text{(B.30)} \\
\beta_3 &= \lfloor \lambda i^s \rfloor \otimes \lfloor \lambda a^s \rfloor \otimes 2 \cdot \lfloor \lambda i^p \rfloor \oplus \beta_2 \otimes 2 \cdot \lfloor \lambda i^p \rfloor \oplus \\
&\quad \beta_3 \otimes \lfloor \lambda i^s \rfloor \otimes \lfloor \lambda a^s \rfloor \otimes \lfloor \lambda i^p \rfloor \oplus \beta_2 \otimes 2 \cdot (\lfloor \lambda i^s \rfloor \otimes \lfloor \lambda a^s \rfloor) && \text{(B.31)} \\
&= \lfloor \lambda i^s \rfloor \otimes \lfloor \lambda a^s \rfloor \otimes 2 \cdot \lfloor \lambda i^p \rfloor \oplus \beta_2 \otimes 2 \cdot \lfloor \lambda i^p \rfloor && \text{(B.32)}
\end{aligned}$$

In equation B.29 and in equation B.31 the terms containing respectively β_2 and β_3 can be removed. In both equations the next propery is used for simplifying the equation:
$\lfloor \lambda i^s \rfloor = \lfloor \lambda i^p \rfloor$ and $\lfloor \lambda a^s \rfloor \leq 0 \Rightarrow \beta_n \otimes 2 \cdot (\lfloor \lambda i^s \rfloor \otimes \lfloor \lambda a^s \rfloor) < \beta_n \otimes 2 \cdot \lfloor \lambda i^p \rfloor$.

First β_2 is substituted in equation B.31. The new value of β_3 is then substituted in equation B.29. Then β_2 and β_3 are substituted in equations B.28 and B.21.

$$\begin{aligned}
\beta_3 &= \lfloor \lambda i^s \rfloor \otimes \lfloor \lambda a^s \rfloor \otimes 2 \cdot \lfloor \lambda i^p \rfloor \oplus 2 \cdot (\lfloor \lambda i^s \rfloor \otimes \lfloor \lambda a^s \rfloor \otimes \lfloor \lambda i^p \rfloor) \oplus \\
&\quad \beta_3 \otimes 4 \cdot \lfloor \lambda i^p \rfloor && \text{(B.33)} \\
&= \lfloor \lambda i^s \rfloor \otimes \lfloor \lambda a^s \rfloor \otimes 2 \cdot \lfloor \lambda i^p \rfloor && \text{(B.34)} \\
\beta_2 &= 2 \cdot (\lfloor \lambda i^s \rfloor \otimes \lfloor \lambda a^s \rfloor) \otimes \lfloor \lambda i^p \rfloor \oplus \lfloor \lambda i^s \rfloor \otimes \lfloor \lambda a^s \rfloor \otimes 4 \cdot \lfloor \lambda i^p \rfloor && \text{(B.35)} \\
\beta_4 &= 2 \cdot (\lfloor \lambda i^s \rfloor \otimes \lfloor \lambda a^s \rfloor \otimes \lfloor \lambda i^p \rfloor) \oplus \lfloor \lambda i^s \rfloor \otimes \lfloor \lambda a^s \rfloor \otimes 5 \cdot \lfloor \lambda i^p \rfloor \oplus \\
&\quad 2 \cdot (\lfloor \lambda i^s \rfloor \otimes \lfloor \lambda a^s \rfloor \otimes \lfloor \lambda i^p \rfloor) && \text{(B.36)} \\
\beta_5 &= 3 \cdot (\lfloor \lambda i^s \rfloor \otimes \lfloor \lambda a^s \rfloor) \otimes \lfloor \lambda i^p \rfloor \oplus 2 \cdot (\lfloor \lambda i^s \rfloor \otimes \lfloor \lambda a^s \rfloor) \otimes 4 \cdot \lfloor \lambda i^p \rfloor \oplus \\
&\quad \lfloor \lambda i^s \rfloor \otimes \lfloor \lambda a^s \rfloor \otimes 3 \cdot \lfloor \lambda i^p \rfloor && \text{(B.37)}
\end{aligned}$$

Summarising and filling in the last steps results in:

$$\begin{aligned}
\beta_0 &= 0 && \text{(B.38)}\\
\beta_1 &= \lfloor \lambda i^s \rfloor \otimes \lfloor \lambda a^s \rfloor \otimes \lfloor \lambda i^p \rfloor && \text{(B.39)}\\
\beta_2 &= 2 \cdot (\lfloor \lambda i^s \rfloor \otimes \lfloor \lambda a^s \rfloor) \otimes \lfloor \lambda i^p \rfloor \oplus \lfloor \lambda i^s \rfloor \otimes \lfloor \lambda a^s \rfloor \otimes 4 \cdot \lfloor \lambda i^p \rfloor && \text{(B.40)}\\
\beta_3 &= \lfloor \lambda i^s \rfloor \otimes \lfloor \lambda a^s \rfloor \otimes 2 \cdot \lfloor \lambda i^p \rfloor && \text{(B.41)}\\
\beta_4 &= 2 \cdot (\lfloor \lambda i^s \rfloor \otimes \lfloor \lambda a^s \rfloor \otimes \lfloor \lambda i^p \rfloor) \oplus \lfloor \lambda i^s \rfloor \otimes \lfloor \lambda a^s \rfloor \otimes 5 \cdot \lfloor \lambda i^p \rfloor && \text{(B.42)}\\
\beta_5 &= 3 \cdot (\lfloor \lambda i^s \rfloor \otimes \lfloor \lambda a^s \rfloor) \otimes \lfloor \lambda i^p \rfloor \oplus \lfloor \lambda i^s \rfloor \otimes \lfloor \lambda a^s \rfloor \otimes 3 \cdot \lfloor \lambda i^p \rfloor && \text{(B.43)}\\
\beta_6 &= \lfloor \lambda i^s \rfloor \otimes \lfloor \lambda a^s \rfloor \otimes 2 \cdot \lfloor \lambda i^p \rfloor && \text{(B.44)}\\
\beta_7 &= \lfloor \lambda i^s \rfloor \otimes \lfloor \lambda a^s \rfloor \otimes 2 \cdot \lfloor \lambda i^p \rfloor && \text{(B.45)}
\end{aligned}$$

In the first paragraph it is mentioned that each eigen-vector contains the same state metric values. The result observed can be written in the next equation:

$$eigenvector(m) = \begin{pmatrix}
\beta_0 & \beta_1 & \beta_2 & \beta_3 & \beta_4 & \beta_5 & \beta_6 & \beta_7 \\
\beta_1 & \beta_0 & \beta_3 & \beta_2 & \beta_5 & \beta_4 & \beta_7 & \beta_6 \\
\beta_2 & \beta_3 & \beta_0 & \beta_1 & \beta_6 & \beta_7 & \beta_4 & \beta_5 \\
\beta_3 & \beta_2 & \beta_1 & \beta_0 & \beta_7 & \beta_6 & \beta_5 & \beta_4 \\
\beta_4 & \beta_5 & \beta_6 & \beta_7 & \beta_0 & \beta_1 & \beta_2 & \beta_3 \\
\beta_5 & \beta_4 & \beta_7 & \beta_6 & \beta_1 & \beta_0 & \beta_3 & \beta_2 \\
\beta_6 & \beta_7 & \beta_4 & \beta_5 & \beta_2 & \beta_3 & \beta_0 & \beta_1 \\
\beta_7 & \beta_6 & \beta_5 & \beta_4 & \beta_3 & \beta_2 & \beta_1 & \beta_0
\end{pmatrix} \cdot \begin{pmatrix}
\delta_{0,m} \\ \delta_{1,m} \\ \delta_{2,m} \\ \delta_{3,m} \\ \delta_{4,m} \\ \delta_{5,m} \\ \delta_{6,m} \\ \delta_{7,m}
\end{pmatrix}$$

Where m is the number of the state with the highest state metric and

$$\delta_{i,j} = \begin{cases} 1 & \text{if i=j} \\ 0 & \text{if i} \neq \text{j} \end{cases}$$

For the forward recursion (α) the matrix is filled the same way. The equations are:

$$\begin{aligned}
\alpha_0 &= 0 && \text{(B.46)}\\
\alpha_1 &= 2 \cdot (\lfloor \lambda i^s \rfloor \otimes \lfloor \lambda a^s \rfloor \otimes \lfloor \lambda i^p \rfloor) \oplus \lfloor \lambda i^s \rfloor \otimes \lfloor \lambda a^s \rfloor \otimes 5 \cdot \lfloor \lambda i^p \rfloor && \text{(B.47)}\\
\alpha_2 &= \lfloor \lambda i^s \rfloor \otimes \lfloor \lambda a^s \rfloor \otimes 2 \cdot \lfloor \lambda i^p \rfloor && \text{(B.48)}\\
\alpha_3 &= 2 \cdot (\lfloor \lambda i^s \rfloor \otimes \lfloor \lambda a^s \rfloor) \otimes \lfloor \lambda i^p \rfloor \oplus \lfloor \lambda i^s \rfloor \otimes \lfloor \lambda a^s \rfloor \otimes 4 \cdot \lfloor \lambda i^p \rfloor && \text{(B.49)}\\
\alpha_4 &= \lfloor \lambda i^s \rfloor \otimes \lfloor \lambda a^s \rfloor \otimes \lfloor \lambda i^p \rfloor && \text{(B.50)}\\
\alpha_5 &= 3 \cdot (\lfloor \lambda i^s \rfloor \otimes \lfloor \lambda a^s \rfloor) \otimes \lfloor \lambda i^p \rfloor \oplus \lfloor \lambda i^s \rfloor \otimes \lfloor \lambda a^s \rfloor \otimes 3 \cdot \lfloor \lambda i^p \rfloor && \text{(B.51)}\\
\alpha_6 &= 2 \cdot (\lfloor \lambda i^s \rfloor \otimes \lfloor \lambda a^s \rfloor) \otimes \lfloor \lambda i^p \rfloor \oplus \lfloor \lambda i^s \rfloor \otimes \lfloor \lambda a^s \rfloor \otimes 4 \cdot \lfloor \lambda i^p \rfloor && \text{(B.52)}\\
\alpha_7 &= 2 \cdot (\lfloor \lambda i^s \rfloor \otimes \lfloor \lambda a^s \rfloor) \otimes \lfloor \lambda i^p \rfloor \oplus \lfloor \lambda i^s \rfloor \otimes \lfloor \lambda a^s \rfloor \otimes 4 \cdot \lfloor \lambda i^p \rfloor && \text{(B.53)}\\
& && \text{(B.54)}
\end{aligned}$$

Note that the equations are the same. Only the assignment differs. As in the β calculations, $\alpha_7 = \alpha_6 = \alpha_3$.

Appendix C

PHIDEO

C.1 Phideo architecture template

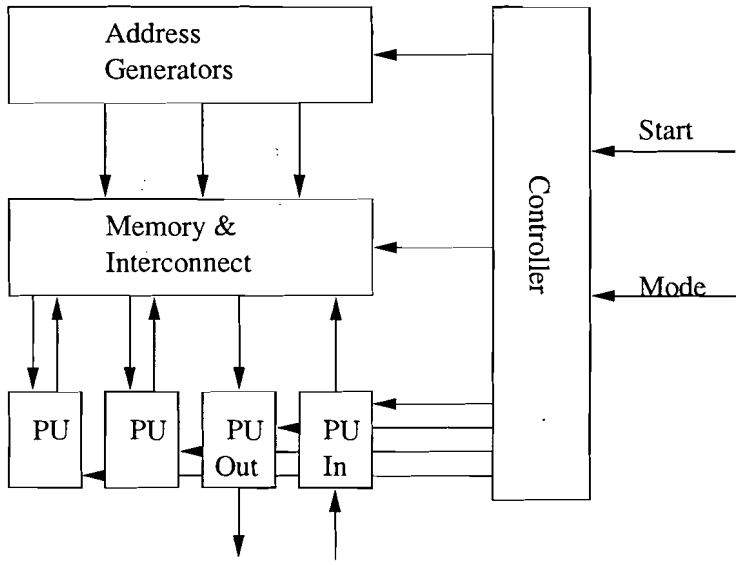

Figure C.1: PHIDEO architecture template

C.2 PIF input

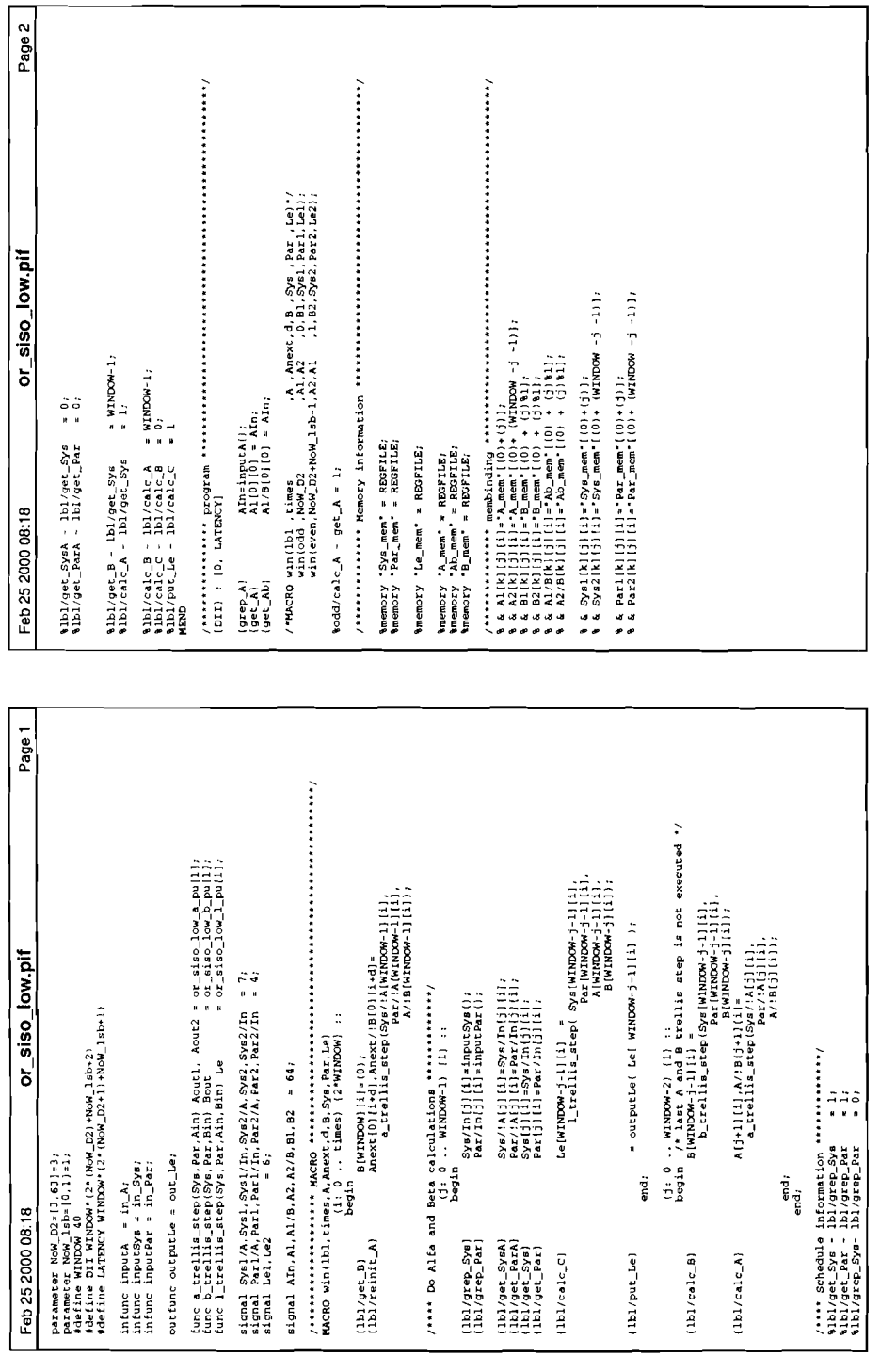

Figure C.2: PIF description of SISO-module

C.3 timing and scheduling

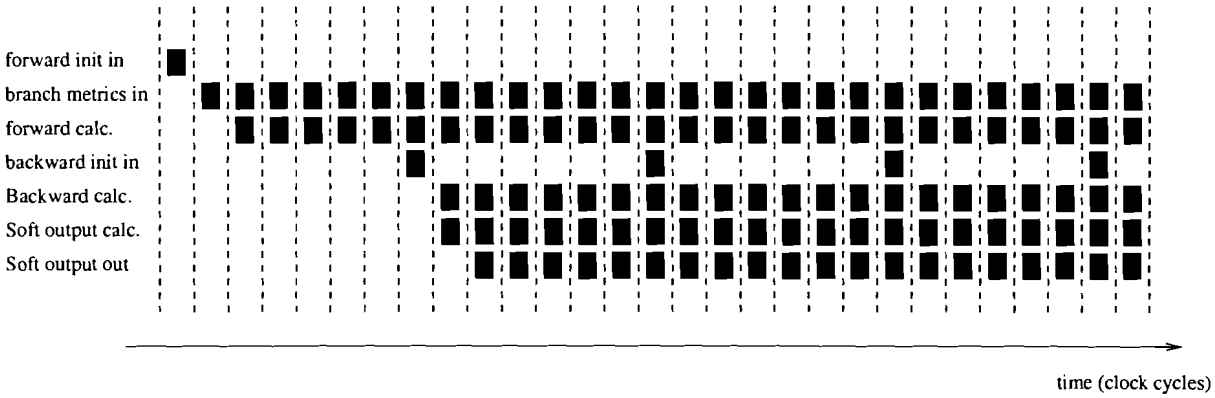

Figure C.3: task schedule

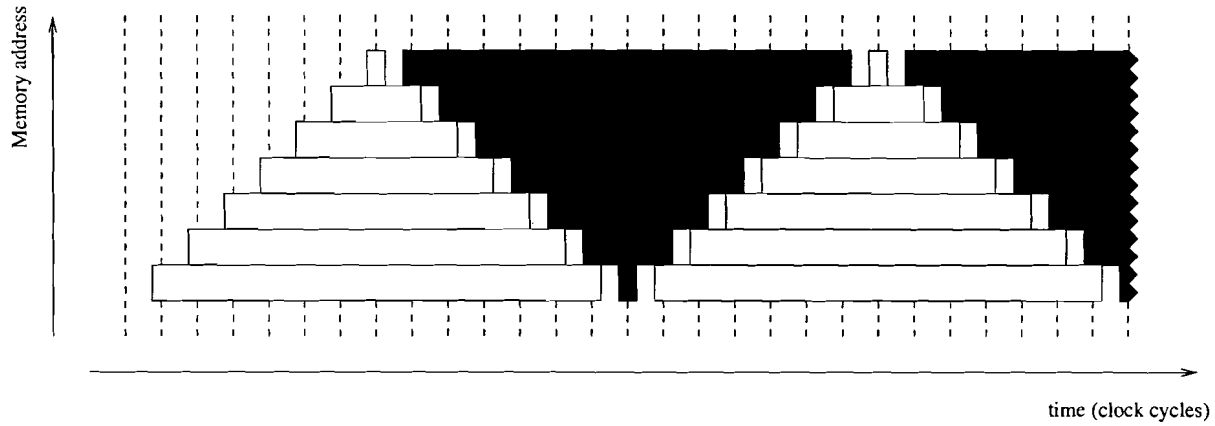

Figure C.4: memory schedule

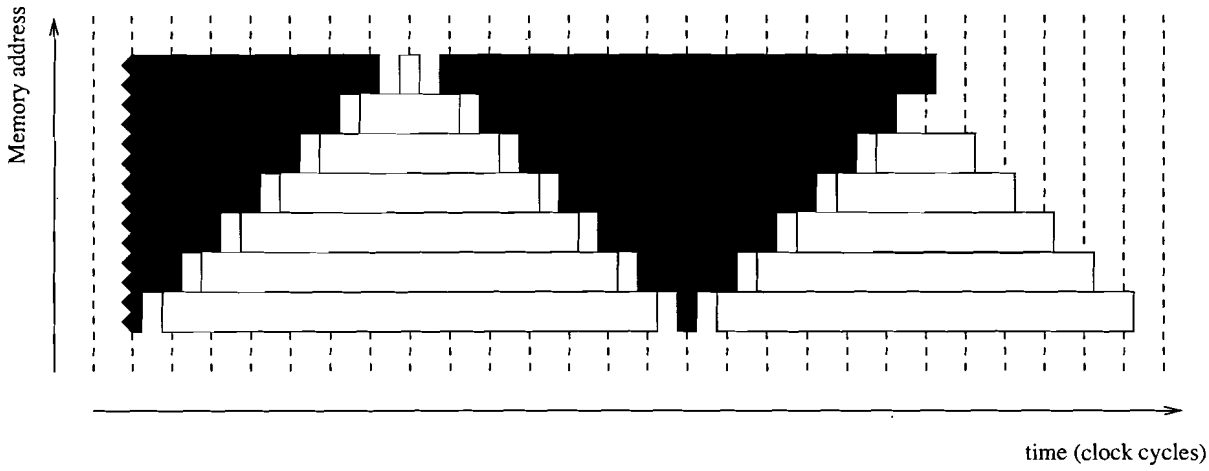

Figure C.5: memory schedule for last window

Appendix D

Data path descriptions

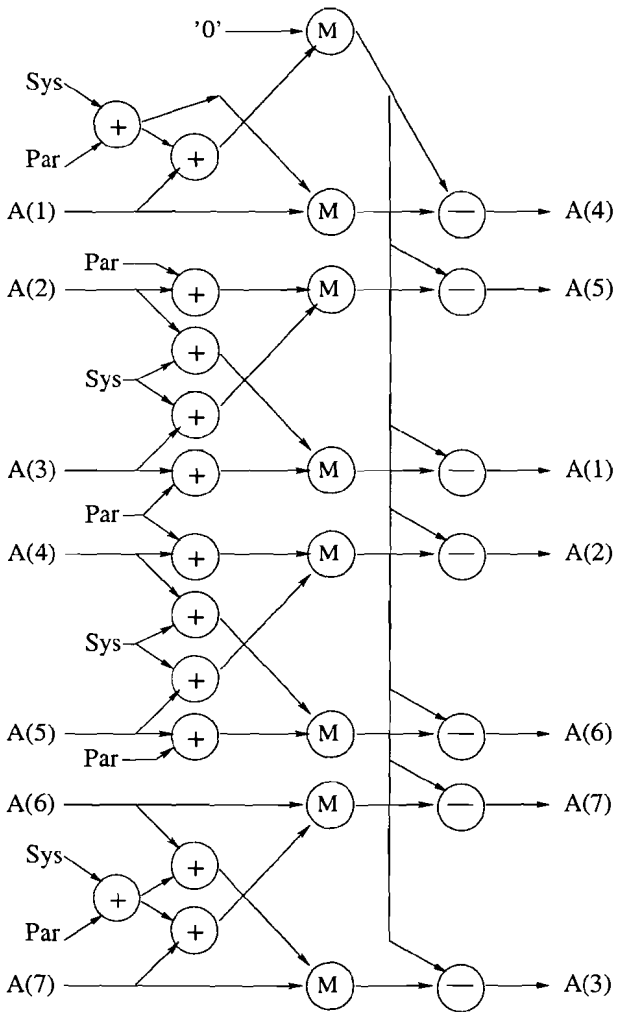

Figure D.1: α data path visualisation

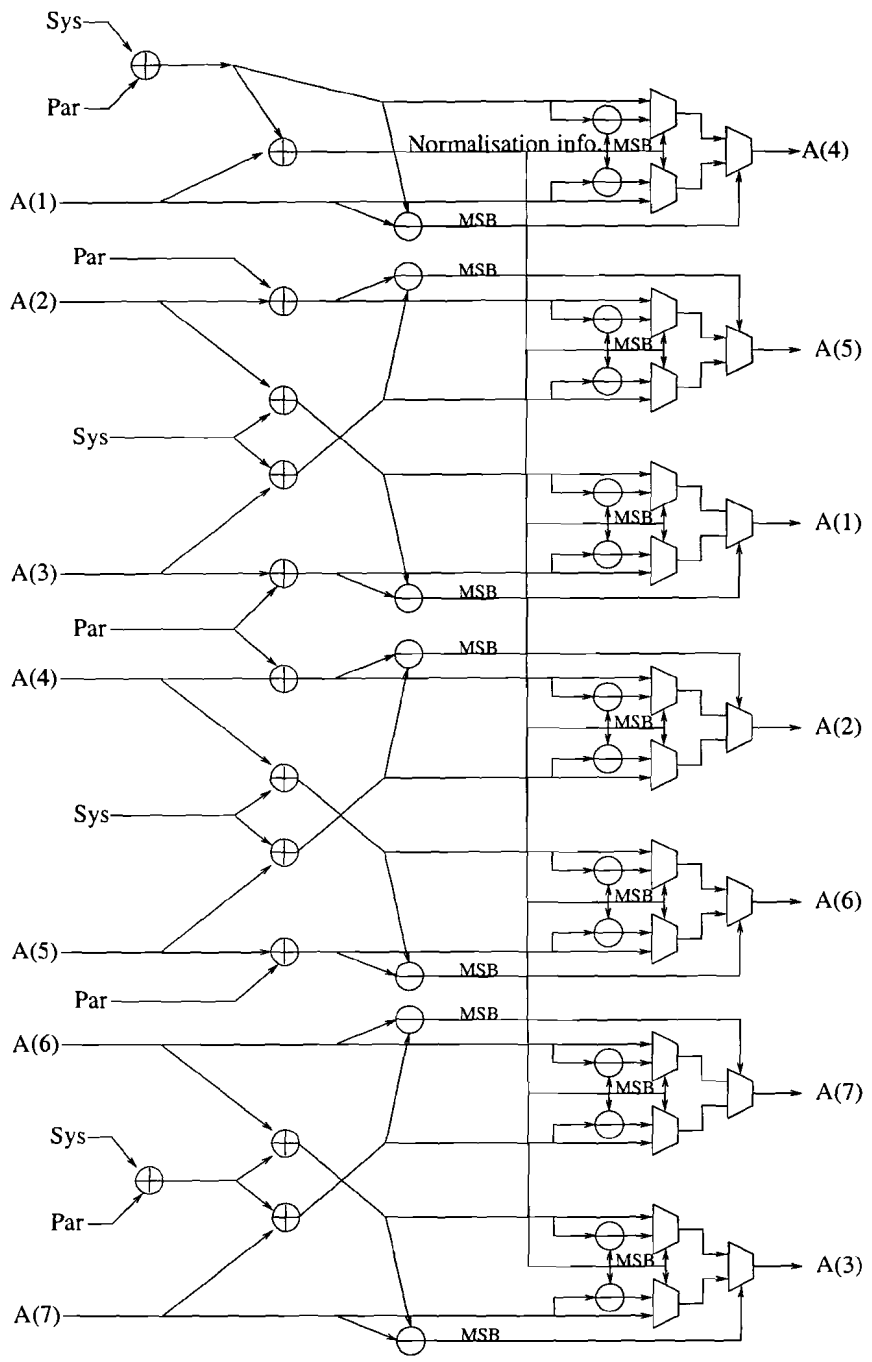

Figure D.2: low latency α data path visualisation

Apr 20 2000 10:05 **or_siso_low_a_pu.behv.a.vhd** Page 1

```
LIBRARY ieee;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;
LIBRARY work;
USE work.or_siso_pack.ALL;

ARCHITECTURE behaviour of or_siso_low_a_pu IS
BEGIN
    proc: PROCESS(Sys, Par, Ain)
        VARIABLE X,Y : signed(N DOWNTO 0);
        VARIABLE s,p : signed(N DOWNTO 0);
        variable tmp : std_logic_vector(N DOWNTO 0);
        VARIABLE A0  : signed(N DOWNTO 0);

    BEGIN

        s := signed(Sys(6)&Sys(6)&Sys(6 DOWNTO 0));
        p := signed(Par(3)&Par(3)&Par(3)&Par(3)&Par(3)&Par(3 DOWNTO 0));

        X := signed(Ain((N)-1) & Ain(N-1 DOWNTO 0));
        Y := signed(Ain((2*N)-1) & Ain((2*N)-1 DOWNTO N)) + s + p;
        A0(N DOWNTO 0):= signed(maxs(X,Y));
        Aout1(N-1 DOWNTO 0) <= "00000000";
        Aout2(N-1 DOWNTO 0) <= "00000000";

        X := signed(Ain((N)-1) & Ain(N-1 DOWNTO 0)) + s + p;
        Y := signed(Ain((2*N)-1) & Ain((2*N)-1 DOWNTO N));
        tmp:= std_logic_vector(maxs(X,Y)-A0);
        Aout1((5*N)-1 DOWNTO 4*N) <= tmp(N-1 DOWNTO 0);
        Aout2((5*N)-1 DOWNTO 4*N) <= tmp(N-1 DOWNTO 0);

        X := signed(Ain((3*N)-1) & Ain((3*N)-1 DOWNTO 2*N)) + s;
        Y := signed(Ain((4*N)-1) & Ain((4*N)-1 DOWNTO 3*N)) + p;
        tmp:= std_logic_vector(maxs(X,Y)-A0);
        Aout1((2*N)-1 DOWNTO N) <= tmp(N-1 DOWNTO 0);
        Aout2((2*N)-1 DOWNTO N) <= tmp(N-1 DOWNTO 0);
        X := signed(Ain((3*N)-1) & Ain((3*N)-1 DOWNTO 2*N)) + p;
        Y := signed(Ain((4*N)-1) & Ain((4*N)-1 DOWNTO 3*N)) + s;
        tmp:= std_logic_vector(maxs(X,Y)-A0);
        Aout1((6*N)-1 DOWNTO 5*N) <= tmp(N-1 DOWNTO 0);
        Aout2((6*N)-1 DOWNTO 5*N) <= tmp(N-1 DOWNTO 0);

        X := signed(Ain((5*N)-1) & Ain((5*N)-1 DOWNTO 4*N)) + p;
        Y := signed(Ain((6*N)-1) & Ain((6*N)-1 DOWNTO 5*N)) + s;
        tmp:= std_logic_vector(maxs(X,Y)-A0);
        Aout1((3*N)-1 DOWNTO 2*N) <= tmp(N-1 DOWNTO 0);
        Aout2((3*N)-1 DOWNTO 2*N) <= tmp(N-1 DOWNTO 0);
        X := signed(Ain((5*N)-1) & Ain((5*N)-1 DOWNTO 4*N)) + s;
        Y := signed(Ain((6*N)-1) & Ain((6*N)-1 DOWNTO 5*N)) + p;
        tmp:= std_logic_vector(maxs(X,Y)-A0);
        Aout1((7*N)-1 DOWNTO 6*N) <= tmp(N-1 DOWNTO 0);
        Aout2((7*N)-1 DOWNTO 6*N) <= tmp(N-1 DOWNTO 0);

        X := signed(Ain((7*N)-1) & Ain((7*N)-1 DOWNTO 6*N)) + s + p;
        Y := signed(Ain((8*N)-1) & Ain((8*N)-1 DOWNTO 7*N));
        tmp:= std_logic_vector(maxs(X,Y)-A0);
        Aout1((4*N)-1 DOWNTO 3*N) <= tmp(N-1 DOWNTO 0);
        Aout2((4*N)-1 DOWNTO 3*N) <= tmp(N-1 DOWNTO 0);
        X := signed(Ain((7*N)-1) & Ain((7*N)-1 DOWNTO 6*N)),
        Y := signed(Ain((8*N)-1) & Ain((8*N)-1 DOWNTO 7*N)) + s + p;
        tmp:= std_logic_vector(maxs(X,Y)-A0);
        Aout1((8*N)-1 DOWNTO 7*N) <= tmp(N-1 DOWNTO 0);
        Aout2((8*N)-1 DOWNTO 7*N) <= tmp(N-1 DOWNTO 0);


    END PROCESS;
END;
```

Apr 20 2000 10:03 **or_siso_low_b_pu.behv.a.vhd** Page 1

```
LIBRARY ieee;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;
LIBRARY work;
USE work.or_siso_pack.ALL;

ARCHITECTURE behaviour OF or_siso_low_b_pu IS
BEGIN
    proc: PROCESS(Sys, Par, Bin)
        VARIABLE X,Y : signed(N DOWNTO 0);
        VARIABLE B0  : signed(N DOWNTO 0);
        variable tmp : std_logic_vector(N DOWNTO 0);
        VARIABLE s,p : signed(N DOWNTO 0);
    BEGIN
--      WAIT ON Sys, Par, Bin;

        s := signed(Sys(6)&Sys(6)&Sys(6 DOWNTO 0));
        p := signed(Par(3)&Par(3)&Par(3)&Par(3)&Par(3)&Par(3 DOWNTO 0));

        X := signed(Bin((N)-1) & Bin(N-1 DOWNTO 0));
        Y := signed(Bin((5*N)-1) & Bin((5*N)-1 DOWNTO 4*N)) + s + p;
        B0(N DOWNTO 0) := signed(maxs(X,Y));
        Bout(N-1 downto 0) <= "00000000";
        X := signed(Bin((N)-1) & Bin(N-1 DOWNTO 0)) + s + p;
        Y := signed(Bin((5*N)-1) & Bin((5*N)-1 DOWNTO 4*N));
        tmp:= std_logic_vector(maxs(X,Y)-B0);
        Bout((2*N)-1 DOWNTO N) <= tmp(N-1 DOWNTO 0);

        X := signed(Bin((2*N)-1) & Bin((2*N)-1 DOWNTO 1*N)) + s;
        Y := signed(Bin((6*N)-1) & Bin((6*N)-1 DOWNTO 5*N)) + p;
        tmp:= std_logic_vector(maxs(X,Y)-B0);
        Bout((3*N)-1 DOWNTO 2*N) <= tmp(N-1 DOWNTO 0);
        X := signed(Bin((2*N)-1) & Bin((2*N)-1 DOWNTO 1*N)) + p;
        Y := signed(Bin((6*N)-1) & Bin((6*N)-1 DOWNTO 5*N)) + s;
        tmp:= std_logic_vector(maxs(X,Y)-B0);
        Bout((4*N)-1 DOWNTO 3*N) <= tmp(N-1 DOWNTO 0);

        X := signed(Bin((3*N)-1) & Bin((3*N)-1 DOWNTO 2*N)) + p;
        Y := signed(Bin((7*N)-1) & Bin((7*N)-1 DOWNTO 6*N)) + s;
        tmp:= std_logic_vector(maxs(X,Y)-B0);
        Bout((5*N)-1 DOWNTO 4*N) <= tmp(N-1 DOWNTO 0);
        X := signed(Bin((3*N)-1) & Bin((3*N)-1 DOWNTO 2*N)) + s;
        Y := signed(Bin((7*N)-1) & Bin((7*N)-1 DOWNTO 6*N)) + p;
        tmp:= std_logic_vector(maxs(X,Y)-B0);
        Bout((6*N)-1 DOWNTO 5*N) <= tmp(N-1 DOWNTO 0);

        X := signed(Bin((4*N)-1) & Bin((4*N)-1 DOWNTO 3*N)) + s + p;
        Y := signed(Bin((8*N)-1) & Bin((8*N)-1 DOWNTO 7*N));
        tmp:= std_logic_vector(maxs(X,Y)-B0);
        Bout((7*N)-1 DOWNTO 6*N) <= tmp(N-1 DOWNTO 0);
        X := signed(Bin((4*N)-1) & Bin((4*N)-1 DOWNTO 3*N));
        Y := signed(Bin((8*N)-1) & Bin((8*N)-1 DOWNTO 7*N)) + s + p;
        tmp:= std_logic_vector(maxs(X,Y)-B0);
        Bout((8*N)-1 DOWNTO 7*N) <= tmp(N-1 DOWNTO 0);

    END PROCESS;
END;
```


Figure D.3: α and β data path description

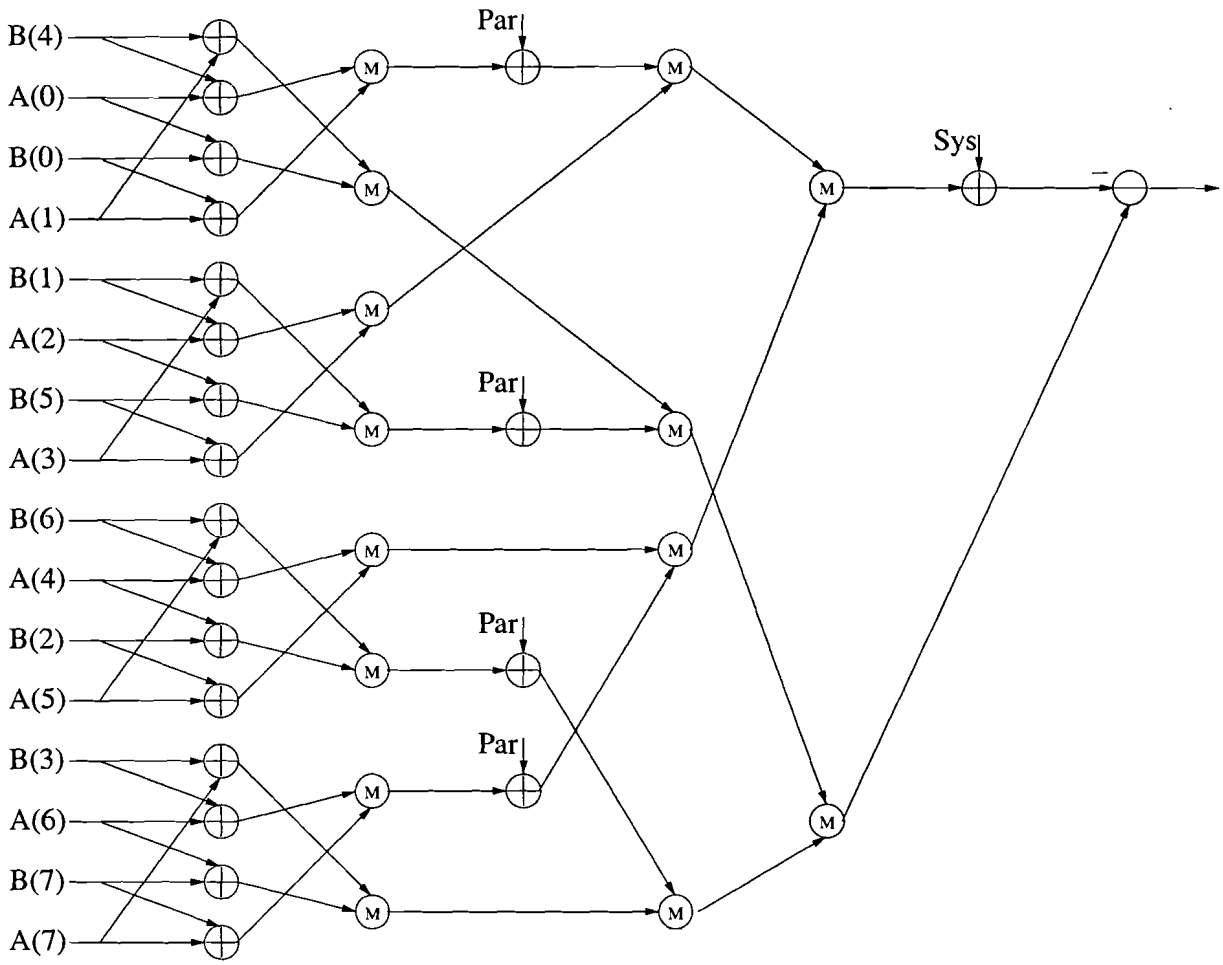

Figure D.4: efficient Λ data path visualisation

Apr 20 2000 10:03 **or_siso_low_l_pu.behv.a.vhd** Page 1

```
LIBRARY ieee;
USE IEEE.std_logic_1164.ALL;
USE IEEE.numeric_std.ALL;
LIBRARY work;
USE work.or_siso_pack.ALL;

ARCHITECTURE behaviour of or_siso_low_l_pu IS
BEGIN
    proc: PROCESS(Sys, Par, Bin, Ain)
        VARIABLE a1,a2,a3,a4,a5,a6,a7,a8,a9,a10,a11,a12,a13,a14,a15,a16  : signed(N DOWN
TO 0);
        VARIABLE b1,b2,b3,b4,b5,b6,b7,b8  : signed(N DOWNTO 0);
        VARIABLE c1,c4,c6,c7              : signed(N DOWNTO 0);
        VARIABLE d1,d2,d3,d4              : signed(N DOWNTO 0);
        VARIABLE e1,e2                    : signed(N DOWNTO 0);
        VARIABLE f                        : signed(N DOWNTO 0);
        VARIABLE s,p                      : signed(N DOWNTO 0);
    BEGIN
--      WAIT UNTIL rising_edge(ck);
--      WAIT ON Sys, Par, Bin, Ain;


-- assuming N=8;
        s := signed(Sys(6)&Sys(6)&Sys(6 DOWNTO 0));
        p := signed(Par(3)&Par(3)&Par(3)&Par(3)&Par(3)&Par(3 DOWNTO 0));

--- stage 1
        a1  := signed((Ain((2*N)-1) & Ain((2*N)-1 DOWNTO N)))    + signed((Bin((5*N)-1) &
 Bin((5*N)-1 DOWNTO 4*N)));
        a2  := signed((Ain((N)-1)   & Ain(N-1 DOWNTO 0)))        + signed((Bin((5*N)-1) &
 Bin((5*N)-1 DOWNTO 4*N)));
        a3  := signed((Ain((N)-1)   & Ain(N-1 DOWNTO 0)))        + signed((Bin((N)-1) & B
in(N-1 DOWNTO 0)));
        a4  := signed((Ain((2*N)-1) & Ain((2*N)-1 DOWNTO N)))    + signed((Bin((N)-1) & B
in(N-1 DOWNTO 0)));

        a5  := signed((Ain((4*N)-1) & Ain((4*N)-1 DOWNTO 3*N))) + signed((Bin((2*N)-1) &
 Bin((2*N)-1 DOWNTO N)));
        a6  := signed((Ain((3*N)-1) & Ain((3*N)-1 DOWNTO 2*N))) + signed((Bin((2*N)-1) &
 Bin((2*N)-1 DOWNTO N)));
        a7  := signed((Ain((3*N)-1) & Ain((3*N)-1 DOWNTO 2*N))) + signed((Bin((6*N)-1) &
 Bin((6*N)-1 DOWNTO 5*N)));
        a8  := signed((Ain((4*N)-1) & Ain((4*N)-1 DOWNTO 3*N))) + signed((Bin((6*N)-1) &
 Bin((6*N)-1 DOWNTO 5*N)));

        a9  := signed((Ain((6*N)-1) & Ain((6*N)-1 DOWNTO 5*N))) + signed((Bin((7*N)-1) &
 Bin((7*N)-1 DOWNTO 6*N)));
        a10 := signed((Ain((5*N)-1) & Ain((5*N)-1 DOWNTO 4*N))) + signed((Bin((7*N)-1) &
 Bin((7*N)-1 DOWNTO 6*N)));
        a11 := signed((Ain((5*N)-1) & Ain((5*N)-1 DOWNTO 4*N))) + signed((Bin((3*N)-1) &
 Bin((3*N)-1 DOWNTO 2*N)));
        a12 := signed((Ain((6*N)-1) & Ain((6*N)-1 DOWNTO 5*N))) + signed((Bin((3*N)-1) &
 Bin((3*N)-1 DOWNTO 2*N)));

        a13 := signed((Ain((8*N)-1) & Ain((8*N)-1 DOWNTO 7*N))) + signed((Bin((4*N)-1) &
 Bin((4*N)-1 DOWNTO 3*N)));
        a14 := signed((Ain((7*N)-1) & Ain((7*N)-1 DOWNTO 6*N))) + signed((Bin((4*N)-1) &
 Bin((4*N)-1 DOWNTO 3*N)));
        a15 := signed((Ain((7*N)-1) & Ain((7*N)-1 DOWNTO 6*N))) + signed((Bin((8*N)-1) &
 Bin((8*N)-1 DOWNTO 7*N)));
        a16 := signed((Ain((8*N)-1) & Ain((8*N)-1 DOWNTO 7*N))) + signed((Bin((8*N)-1) &
 Bin((8*N)-1 DOWNTO 7*N)));

--- stage 2

        b1 := maxs(a2,a4);
        b2 := maxs(a1,a3);
        b3 := maxs(a6,a8);
        b4 := maxs(a5,a7);
```

Apr 20 2000 10:03 **or_siso_low_l_pu.behv.a.vhd** Page 2

```
        b5 := maxs(a10,a12);
        b6 := maxs(a9,a11);
        b7 := maxs(a14,a16);
        b8 := maxs(a13,a15);

--- stage 3  ------- assuming that this will not generate an overflow. (should
--- not)

        c1 := b1 + p;
        c4 := b4 + p;
        c6 := b6 + p;
        c7 := b7 + p;

--- stage 4

        d1 := maxs(c1,b3);
        d2 := maxs(b2,c4);
        d3 := maxs(b5,c7);
        d4 := maxs(c6,b8);


--- stage 5

        e1 := maxs(d1,d3);
        e2 := maxs(d2,d4);

--- stage 6

        f := e1 - e2 + s; -- => Le+Ay+Li
        Le <= std_logic_vector(f(5 DOWNTO 0));

    END PROCESS;
END;
```


Figure D.5: Λ data path description