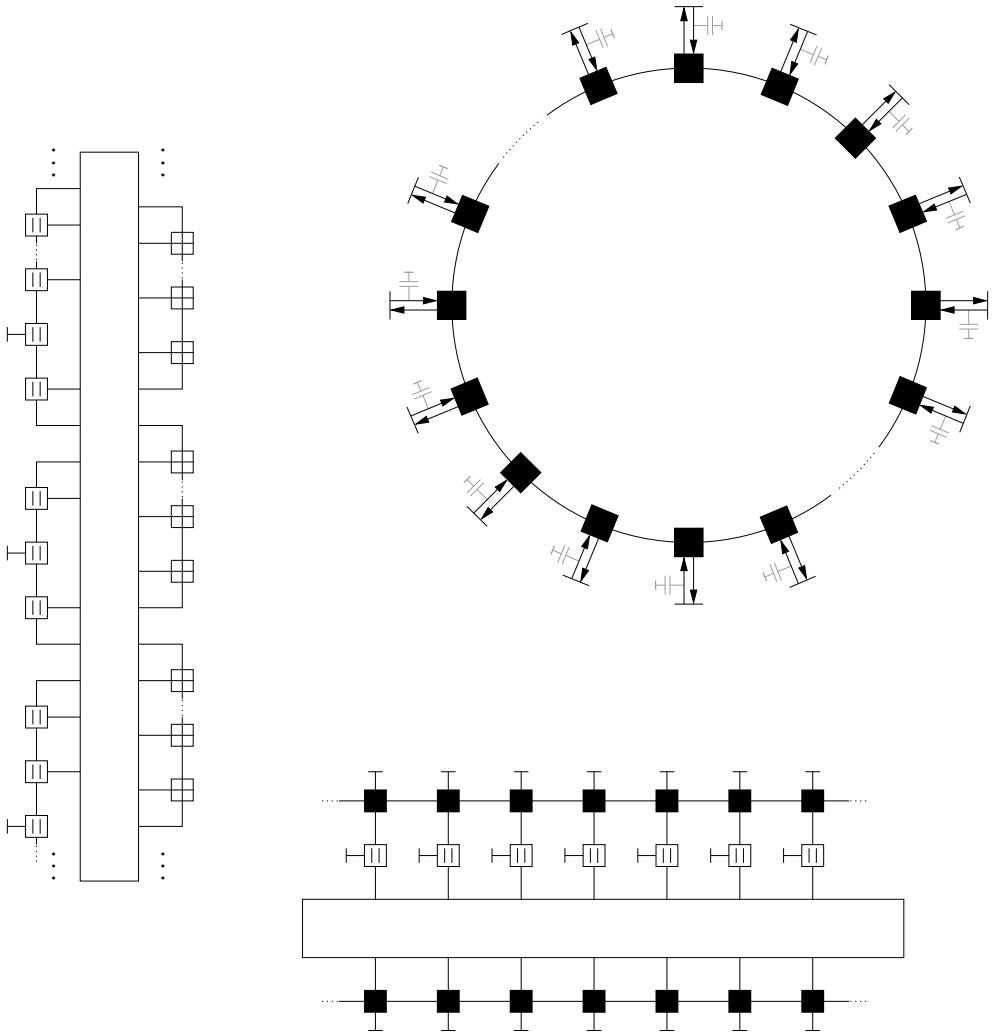
# Analog Signal Processing in Forward Error Correction (FEC) Decoders



**Matthias Mörz**

**Lehrstuhl für Nachrichtentechnik**

# Analog Signal Processing in Forward Error Correction (FEC) Decoders

Matthias Mörz

Vollständiger Abdruck der von der Fakultät für Elektrotechnik und Informationstechnik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktor–Ingenieurs

genehmigten Dissertation.

| | |
|---|---|
| Vorsitzender: | Univ.–Prof. Dr. sc. techn. (ETH) Andreas Herkersdorf |
| Prüfer der Dissertation: | |
| | 1. Univ.–Prof. Dr.–Ing. Dr.–Ing. E. h. Joachim Hagenauer, i.R. |
| | 2. Univ.–Prof. Dr.–Ing. habil. Norbert Wehn, Technische Universität Kaiserslautern |

Die Dissertation wurde am 18.01.2007 bei der Technischen Universität München eingereicht und durch die Fakultät für Elektrotechnik und Informationstechnik am 21.06.2007 angenommen.

# Preface

This thesis is a result of my work as research assistant at the Institute for Communications Engineering (LNT) at the Munich University of Technology (TUM).

First of all, I would like to thank my supervisor Prof. Dr.–Ing. Dr.–Ing. E. h. Joachim Hagenauer for giving me the opportunity to contribute to such an exciting new area of research and all the support he provided throughout the last years. I am especially grateful that he made it possible for me to participate in a joint research project with Bell Labs, Lucent Technologies. I would also like to thank Prof. Dr.–Ing. Norbert Wehn for acting as co-examiner.

At this point, I would also like to express my thanks to Dr. Ran Yan, the former vice president of Wireless Research at Bell Labs, Lucent Technologies, for supporting more than three years of this work. It has been a great pleasure for me to spend some time of my work at Bell Labs in Murray Hill, NJ, in Swindon, the United Kingdom, and in Holmdel, NJ. Within Bell Labs I am in particular indebted to Thad Gabara, Frank Hrycrnko, Ted Gabara, Cyril Measson, Sami Hyvonen and Eric Westerwick for their contributions to the successful chip implementations.

I am also very thankful for the great environment and enjoyable atmosphere at the LNT. I would therefore like to express my sincere appreciation to all my former colleagues and my past diploma, master and bachelor students. All of them contributed to this work in one way or the other. Special thanks to Dr. Andrew Schaefer, James Ghirlando, Pavol Hanus, Andreas Müller, Janis Dingel, Prof. Dr.–Ing. Reimar Lenz and the former system administrators Dr. Stephan Bäro, Dr. Markus Kaindl, Günther Liebl and Dr. Johannes Zangl who always kept the computers up and running. Additional thanks to Dr. Andrew Schaefer and Dr. Reza Karimi for proof-reading parts of my thesis.

Finally, I would like to thank my parents and in particular my wife Silvie for their continuous support and always encouraging me in my work.

München, January 2007 *Matthias Mörz*

To my family

# *Contents*

## Abstract

This work deals with a new type of analog signal processing in the forward error correction (FEC) decoder of a digital communication system. Such analog FEC decoders are studied based on a comprehensive simulation environment including system-level and circuit-level simulation models. Different decoder architectures are considered. This includes fully parallel decoders and decoders based on a new sequential technique for complexity reduction, which is examined for the example of the UMTS turbo code. A library of analog transistor circuits is presented which is suited for the realization of arbitrary analog decoders. As a proof of concept two prototypes of analog decoders were successfully fabricated in 0.25 $\mu$m BiCMOS and 0.25 $\mu$m SiGe. Finally, analog low-density parity-check (LDPC) decoders in 0.18 $\mu$m CMOS are investigated for an application in the next generation wireless local area network IEEE 802.11n.



## Zusammenfassung

Diese Arbeit beschäftigt sich mit einer neuartigen analogen Signalverarbeitung im Kanaldecoder eines digitalen Kommunikationssystems. Zur Untersuchung solcher analoger Kanaldecoder wurde eine Simulationsumgebung entwickelt, welche verschiedene Simulationsmodelle auf System- und Schaltungsebene beinhaltet. Neben parallelen Decoderarchitekturen werden auch neue sequentielle Verfahren zur Komplexitätsreduzierung betrachtet und am Beispiel des Turbocodes für UMTS näher untersucht. Es wird eine Bibliothek von analogen Transistorschaltungen vorgestellt, mit der sich beliebige analoge Decoder realisieren lassen. Das Grundkonzept der analogen Decodierung wurde durch die erfolgreiche Fertigung zweier Prototypen in 0,25 $\mu$m BiCMOS und 0,25 $\mu$m SiGe nachgewiesen. Zum Schluss werden analoge LDPC-Decoder in 0,18 $\mu$m CMOS untersucht, die auf eine Anwendung in zukünftigen drahtlosen Netzwerken nach IEEE 802.11n zielen.

# 1

# Introduction

It is nowadays common practice to transmit and store information of any kind in the form of binary digits, so-called bits. Analog signals as they are naturally available in the real world are therefore converted into a digital signal representation. Digital signals allow the application of powerful signal processing techniques like source coding and data compression so that the redundancy in the signal can be removed. It is then possible to represent analog signals with a minimum number of bits in the digital domain. Another important advantage of digital signals is that so-called forward error correction (FEC) can be applied. FEC is an integral part of almost all digital communication systems in order to guarantee a reliable bit transmission despite the presence of noise and other disturbances in the communication channel. It contributes two additional signal processing blocks to the communication system, an encoder at the transmitter and a channel decoder at the receiver. The encoder adds redundancy to the bit sequence in a well-defined manner. This redundancy can then be exploited by the channel decoder in order to detect and correct transmission errors. The use of FEC is also referred to as channel coding. It not only reduces the number of transmission errors but also facilitates an extended operating range and/or a reduced transmit power. The fundamental limits for channel coding were originally derived by Claude E. Shannon [Sha48] in 1948. He defined the theoretical limit, which he termed channel capacity, for the reliable transmission of information over a given noisy communication channel. Shannon further stated that there exists a FEC scheme which achieves an arbitrary small bit error probability as long as the transmission rate does not exceed the channel capacity. The only hint to the construction of suited channel codes was that the code needs to span a large number of bits. The pioneering work of Shannon triggered a several decade long effort to identify such codes.

In the meantime, several FEC schemes have been identified which closely approach the theoretical limits of data transmission over noisy communication channels. These include low-density parity-check (LDPC) codes [Gal62] and turbo codes [BGT93]. To date, the application of these coding schemes has been mainly limited by the availability of powerful and energy-efficient signal processing in the receiver. The increasing data rates in modern communication systems with hundreds of megabits per second or even gigabits per second make it even more challenging to implement decoders for such FEC schemes by means of digital signal processing. In fact, for many high-speed applications it is nowadays not feasible to implement digital decoders for powerful FEC schemes due to area (i.e., cost) and power constraints. Power consumption is critical for both wireless and wire-line applications because of battery life and/or heat dissipation in highly integrated circuits. Furthermore, processing delay is often constrained by the application. One of the reasons for the complexity of the signal processing in the decoder

is that, in order to unfold its optimal performance, a decoder needs to operate on the real numbers provided by the channel in the form of analog signals. It is one of the lessons learned in communications engineering that a hard decision on the transmitted bits should be avoided in the receiver until the original source signal is being reconstructed. A state-of-the-art digital receiver therefore needs to work with quantized versions of these real numbers in all stages of the receiver, which, for a sufficiently large number of quantization levels then facilitates near-peak performance.

Analog Viterbi decoders were probably the first to address the operations of the channel decoder through analog signal processing, see, e.g., [AG78], [MS93], [SJM94], [SJM98], [DT98], [HC00]. Signal processing in the analog domain has some fundamental advantages compared to conventional digital decoder implementations. Analog decoders are in general more area- and power-efficient than their digital counterparts and require no analog-to-digital conversion at the input. The latter allows us to capture the full error correcting performance of the channel decoder since there is no performance loss due to quantization at the input. Analog Viterbi decoders involve additional digital signal processing and in particular digital memory. They only provide sequences of bits at their output. In the late 1990s, Hagenauer [Hag97b], [Hag98], [HW98] and Loeliger *et al.* [LLHT98] independently proposed new types of purely analog channel decoders which also provide analog outputs for the individual bits. This analog output represents reliability information about the bits which can then be further exploited in subsequent stages of the receiver. These publications triggered a great deal of research activity in this area since this type of analog decoder is naturally suited for the information exchange between different components in the receiver based on the turbo principle [Hag97a]. This information exchange is also a fundamental part of all state-of-the-art coding schemes such as LDPC codes and turbo codes. The corresponding decoders are conventionally realized in the digital domain using an iterative exchange of (essentially analog) information between two or more component decoders. In such a scenario analog decoders are expected to outperform iterative decoders by up to several orders of magnitude in terms of speed, area and/or power consumption.

The pioneering work of Hagenauer and Loeliger *et al.* raised numerous questions in the coding community about the error correcting capabilities of analog decoders. Some of these questions relate to the time-continuous behavior of analog decoders which is in clear contrast to the time-discrete operation of digital decoders. Furthermore, many popular coding schemes also introduce feedback loops into the analog decoding network. *Is analog decoding equivalent to digital decoding or is it better?* Other concerns focus on the imperfectness of analog integrated circuits including noise effects and variations in device size, process and temperature. *What is the effect of these impairments on overall decoder performance?* Further questions originate from a more practical background: *Are analog decoders suited for commercial applications? Can they be integrated together with other digital components of the receiver? What is the overall decoder performance in case there are digital input and output interfaces so that digital-to-analog and analog-to-digital converters are required?* All these questions directly led to first prototype developments by two research groups around Hagenauer and Loeliger. The first successful chip implementations of these groups attracted a lot of attention and many more researchers started to work in this exciting new area. In the meantime analog decoder chips have been fabricated for a variety of different coding schemes and technologies [LHL+99a], [Lus00], [WDK+01], [WDL+01], [XVG+02], [GG03b], [GG03a], [Gau03], [WDY+04], [NWGS04], [ABM+04], [FLL+04], [Win04], [Ama04], [VGN+05], [ABM+05], [FLMS05], [WNGS06], [Arz06] and [HBP06]. The goal of this thesis is to contribute answers to the above questions and beyond.

The thesis is organized as follows: **Chapter 2** provides a short overview of a basic digital communication system. The overall transmission system is then reduced to a time-discrete simulation model with a channel encoder at the transmitter and a channel decoder at the receiver. We summarize different channel codes which are considered for an analog decoder implemen-

tation. **Chapter 3** covers graphical models for the visualization of channel codes. These graphs visualize the structure of the code which is also exploited by the associated channel decoder. In this work we pay particular attention to code graphs introduced by Forney. This is because these code graphs, after some modifications, represent our analog decoding networks in a manner similar to a block diagram. In **Chapter 4** we formulate the general decoding problem and present different decoding algorithms in the context of a general message passing decoding algorithm based on Forney graphs. Both optimal and suboptimal decoding algorithms are considered. We also summarize a powerful technique for the analysis of iterative decoding which we later utilize for the analysis of analog decoders and the optimization of circuit-level parameters thereof. Analog decoders with digital input interfaces necessitate a quantization of the input values which is also investigated here. **Chapter 5** investigates the performance of various different analog decoding networks in terms of the bit error rate. We start with an introduction of our unified simulation environment for analog decoders which includes different time-continuous, time-discrete and circuit-level simulation models. This allows the evaluation of a given decoder or input configuration with different simulation models providing different levels of accuracy. Our simulation environment is also exploited for the development of accurate and fast simulation models at a higher-level which can directly be validated against more detailed simulation models. We highlight different decoder architectures for state-of-the-art turbo codes and LDPC codes. Of particular interest is a novel sequential decoder architecture for turbo codes which circumvents some of the limitations of analog decoders. Furthermore, we also comment on the equivalence between analog and digital decoding. In **Chapter 6** we introduce our circuit design for analog decoding networks. We demonstrate that based on the exponential characteristic of bipolar transistors, the blocks realize the ideal decoder operations. We cover the circuit design of the main building blocks and demonstrate how the building blocks can be interconnected using different interfacing circuits. Different decoder examples are given in order to outline the construction of analog decoders based on our building blocks and the interfacing circuitry. We then analyze the complexity of turbo decoders and LDPC decoders. **Chapter 7** presents some of the highlights of this thesis. We summarize the measurement results of two successfully implemented analog decoder chips which were fabricated in 0.25 $\mu$m BiCMOS and 0.25 $\mu$m SiGe technology, respectively. Both decoders are for a simple tailbiting convolutional code and are intended as proof of concept rather than for a commercial application. Our measurement results include the measured bit error rates as well as an analysis of the dynamic behavior of the decoder chips. To the best of our knowledge, the BiCMOS decoder represents the world's first fully operational analog decoder chip while the SiGe decoder appears to be the fastest analog decoder chip to date. In **Chapter 8** we give an outlook to a commercial application in the emerging wireless local area network (LAN) standard IEEE 802.11n. We investigate the implementation of different analog LDPC decoders in 0.18 $\mu$m CMOS technology. A realization in CMOS is particularly interesting because the transistors do not exhibit an exponential characteristic, and so the operations of the building blocks significantly diverge from the desired behavior. We cover the overall architecture of such a solution, including digital input and output interfaces. Various different impairments of such CMOS implementations are investigated. This includes the error introduced by the building blocks, noise, supply voltage variations, device mismatch and process variations. We estimate the performance and the speed of the analog LDPC decoders for different temperatures and include detailed estimations of transistor count, area and power consumption of the overall LDPC decoder. The thesis concludes with **Chapter 9** where we summarize the main achievements of this work and also indicate directions for further research.

Parts of the material presented in this work have already been published in [HOMM99], [HMO00b], [MGYH00], [HMO00c], [MHO00], [HMO00a], [HMS02], [Moe02], [SSM+03a], [Moe03], [SSM+03b], [Moe04a], [Moe04b], [Moe05] and [Moe06]. Related material which goes beyond the scope of this thesis can be found in [Moe01], [MSO01], [MSOH01].

# 2

# *Fundamentals*

This chapter starts with a brief overview of the basic elements of a digital communication system including the used notation. The overall transmission system is then reduced to a time-discrete simulation model which includes the modulation in the transmitter, the physical transmission channel and the demodulation in the receiver. We pay particular attention to forward error correction (FEC) and give a short overview of different channel codes which are considered in this work. This section is followed by a primer on analog FEC decoders. The chapter concludes with the theoretical limits for channel coding as set out by Claude E. Shannon in 1948.

## 2.1 Digital Communication System

Fig. 2.1 depicts the basic elements of a digital communication system where some arbitrary information is transmitted from the information source to the information sink. The information at the source may be available in the form of an analog or a digital signal. Analog signals need to be transformed into a sequence of time-discrete samples which are represented with a finite number of quantization levels. These samples can then also be treated as digital signal. The task of the source encoder is to represent this information with as few bits as possible. This process is therefore also referred to as data compression. An ideal source encoder removes all redundancy which is typically present in the input signal and generates statistically independent and equally likely output bits. These bits form the sequence of information bits $\{u_k\}$ which is applied to the

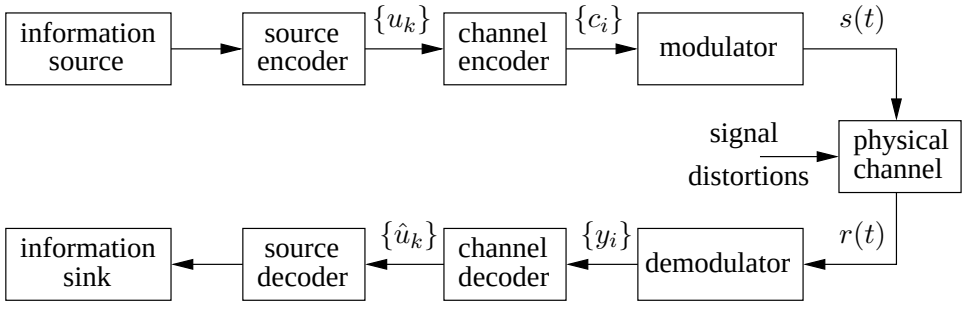



**Figure 2.1**: Block diagram of a digital communication system.

input of the channel encoder. The channel encoder adds a certain amount of redundancy to the information bits in a way that transmission errors which may occur during the signal transmission can be detected and also be corrected in the receiver. In general, more redundancy improves the error correcting capability. The output of the channel encoder is given by the sequence of code bits $\{c_i\}$. In order to transmit this digital information over an essentially analog communication channel we need to map this information onto analog waveforms. This is achieved in the modulator. In this work we restrict ourselves to binary modulation where each code bit $c_i$ is represented by a dedicated analog waveform $s_{c_i}(t)$. The sequence of the analog waveforms $s_{c_i}(t)$ then forms the signal $s(t)$. This signal is transmitted over the physical communication channel which links the transmitter with the receiver. The communication channel is characterized by various different physical phenomena which distort the transmitted signal, such as noise and interference. In the receiver, the demodulator processes the distorted signal $r(t)$ and extracts information about the sequence of code bits in the form of a sequence of samples $\{y_i\}$ for the transmitted symbols. These samples are time-discrete but value-continuous. At this point it is important that the detection of a received symbol, i.e., a hard decision, is avoided in order to preserve the reliability information about the symbols in the receiver [Mas74], [Hag94]. The channel decoder attempts to estimate the sequence of information bits based on the sequence of noisy samples $\{y_i\}$ which include the redundant information added by the channel encoder. The sequence of estimated information bits $\{\hat{u}_k\}$ is then provided to the source decoder which tries to reconstruct the information at the input of the source encoder. This work focuses on the channel decoding part of the communication system in Fig. 2.1 and here in particular on the use of analog signal processing therein. We therefore restrict ourselves to a simplified model of a digital communication system as shown in Fig. 2.2. We assume that there is a digital source which

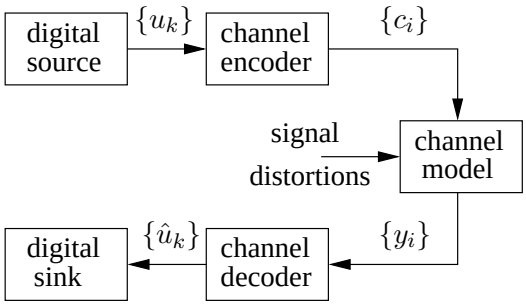



**Figure 2.2**: Simplified model of a digital communication system.

generates a sequence of statistically independent and equally likely information bits $\{u_k\}$. This block replaces the information source and the ideal source encoder in Fig. 2.1. Similarly, a digital sink replaces the source decoder and the information sink. The modulator and demodulator are combined together with the physical communication channel into a mathematical channel model. This channel model reflects the physical phenomena on the communication channel and directly relates the sequence of received samples $\{y_i\}$ to the sequence of code bits $\{c_i\}$. In the following section we focus on a particular channel model as it is assumed throughout this work.

## 2.2 Time-Discrete Channel Model

This section introduces a simple time-discrete channel model as it is commonly used for the simulation and evaluation of channel codes. The channel model in Fig. 2.2 incorporates a modulator, a mathematical description for the characteristics of the communication channel and a demodulator as shown in Fig. 2.3. We restrict ourselves to binary phase shift keying (BPSK) modulation where a binary code symbol $c_i$, $c_i \in \{0, 1\}$, is mapped onto the antipodal ana-

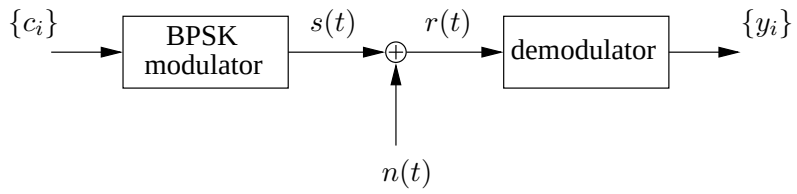



**Figure 2.3**: Simple channel model.

log waveform $s_{c_i}(t)$ with $s_1(t) = -s_0(t)$. Both waveforms are bounded to the corresponding symbol interval $0 < t \leq T_S$. The sequence of waveforms $s_{c_i}(t)$ then yields the transmitted signal $s(t)$. A simple mathematical model for memoryless communication channels is the additive white Gaussian noise (AWGN) channel. This channel model is commonly used for the design, the analysis and the simulation of different communication systems. Here, the received waveform $r(t)$ is distorted according to $r(t) = s(t) + n(t)$ with $n(t)$ as a sample function of a stationary Gaussian process with zero mean and two-sided power spectral density $N_0/2$. We assume that the receiver is perfectly synchronized with the transmitter and fully recovers the phase of the signal. The demodulator then processes the received waveform $r(t)$ in order to provide the sequence of noisy samples $\{y_i\}$. The optimum demodulator for the AWGN channel utilizes a filter which is matched to the pulse shape of the transmitted signal. Such a demodulator is optimal in the sense that it maximizes the signal-to-noise ratio (SNR) at the output of the matched filter at the sampling instant $t = T_S$ [Pro95]. The SNR is then given as

$$\text{SNR} = \frac{2E_S}{N_0}, \tag{2.1}$$

with $E_S$ as the energy per transmitted symbol. The channel model in Fig. 2.3 can then be replaced with the normalized time-discrete AWGN channel model shown in Fig. 2.4. The input to

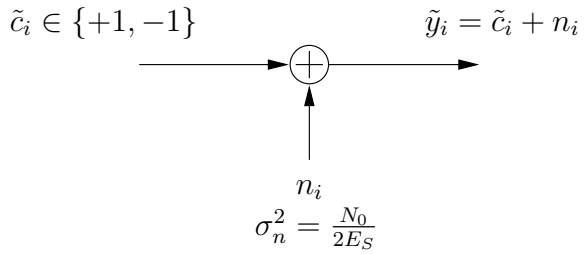



**Figure 2.4**: Normalized AWGN channel model.

this normalized channel model is determined by the modulated code symbols $\tilde{c}_i$, $\tilde{c}_i \in \{+1, -1\}$, which are obtained from the code symbols $c_i$, $c_i \in \{0, 1\}$, through the bijective map $0 \leftrightarrow +1$ and $1 \leftrightarrow -1$. The channel output is given as $\tilde{y}_i = \tilde{c}_i + n_i$ with $n_i$ as a realization of a Gaussian distributed random variable with zero mean and variance

$$\sigma_n^2 = \frac{N_0}{2E_S}. \tag{2.2}$$

Note that the output $\tilde{y}_i$ of the time-discrete channel model in Fig. 2.4 differs from the output $y_i$ in Fig. 2.3 due to the normalization of the channel input, i.e., the signal component, to $\pm 1$. This normalization also affects the noise component so that the SNR in (2.1) remains unchanged.

In order to allow a fair comparison between a communication system with and without channel coding we assume that both systems have the same amount of energy available for the

transmission of the information bits. For a coded transmission system where a sequence of $K$ information bits is encoded into a sequence of $N$ code bits ($N > K$) this implies that the energy divides equally among a larger number of code bits. We therefore obtain for the energy of one transmitted symbol

$$E_S = \frac{K}{N} E_b = R E_b, \tag{2.3}$$

with $E_b$ as the energy per information bit and $R = K/N$ as the rate of the code.

The distribution of the channel output $\tilde{y}_i$ can be described by the conditional probability density functions (pdfs)

$$p(\tilde{y}_i|\tilde{c}_i = \pm 1) = \frac{1}{\sqrt{2\pi\sigma_n^2}} e^{-\frac{(\tilde{y}_i \mp 1)^2}{2\sigma_n^2}}, \tag{2.4}$$

with mean $m_{\tilde{y}} = \tilde{c}_i = \pm 1$ and variance $\sigma_{\tilde{y}}^2 = \sigma_n^2 = N_0/2E_S$. For a simple detector which only provides the hard decisions for the transmitted symbols it suffices to detect the sign of $\tilde{y}_i$. In case of $\tilde{y}_i \geq 0$ we obtain $\tilde{c}_i = +1$ and the output is $\tilde{c}_i = -1$ for $\tilde{y}_i < 0$. With the inverse mapping used in the modulator we then obtain the corresponding code bit $c_i$. The probability of a transmission error on the AWGN channel is determined by

$$P_b = \frac{1}{2} \operatorname{erfc}\left(\frac{m_{\tilde{y}}}{\sqrt{2\sigma_{\tilde{y}}^2}}\right) = \frac{1}{2} \operatorname{erfc}\left(\sqrt{\frac{E_S}{N_0}}\right), \tag{2.5}$$

with

$$\operatorname{erfc}(x) = \frac{2}{\sqrt{\pi}} \int_x^\infty e^{-t^2} dt \tag{2.6}$$

as the complementary error function [Pro95].

The soft output of the AWGN channel is conveniently characterized in terms of the log-likelihood ratio [Hag94]

$$L(\tilde{y}_i|\tilde{c}_i) = \ln \frac{p(\tilde{y}_i|\tilde{c}_i = +1)}{p(\tilde{y}_i|\tilde{c}_i = -1)} = \ln \frac{e^{-\frac{(\tilde{y}_i - 1)^2}{2\sigma_n^2}}}{e^{-\frac{(\tilde{y}_i + 1)^2}{2\sigma_n^2}}} \tag{2.7}$$

$$= 4\frac{E_S}{N_0} \tilde{y}_i = L_c \tilde{y}_i, \tag{2.8}$$

where $\ln$ denotes the natural logarithm. The factor $L_c = 4E_S/N_0$ in (2.8) represents the channel state information (CSI) which acts as a SNR dependent measure for the reliability of the communication channel. This log-likelihood ratio is also referred to as L-value obtained from the channel.

All of the following considerations of this work are based on the normalized and time-discrete AWGN channel model in Fig. 2.3. It is implicitly understood that code bits $c_i$, $c_i \in \{0, 1\}$, are mapped onto the symbols $\tilde{c}_i$, $\tilde{c}_i \in \{+1, -1\}$, before transmission over the AWGN channel. This step is therefore omitted for simplicity. Also, the output $\tilde{y}_i$ of the normalized AWGN channel model is simply referred to as matched filter output $y_i$.

## 2.3 Forward Error Correction (FEC)

Forward error correction (FEC) codes are nowadays commonly used in order to mitigate the effects of noise and interference encountered during the transmission of signals over a communication channel. FEC codes are therefore also referred to as channel codes. Popular applications include digital video broadcast (DVB), wireless LANs and mobile phones. The main goal of channel coding is to achieve a reliable communication at data transmission rates close to the capacity of the communication channel [Sha48]. Channel codes can be grouped into the class of block and convolutional codes depending on whether the information bits are encoded one block at a time or in a more continuous fashion. A special case are tailbiting convolutional codes [SvT79], [MW86] which can also be treated as block codes. In 1966 Forney demonstrated that powerful codes can be generated based on a concatenation of simple component codes [For66]. The key advantage of this approach is that sequential processing of the component codes in the receiver has only a moderate complexity compared to processing the overall code in one step. Serially concatenated codes found their first application in the area of deep space communications [CHIW98]. A major breakthrough was achieved with the parallel concatenation of recursive systematic convolutional codes by Berrou *et al.* in 1993 [BGT93]. This class of codes was termed turbo codes. The discovery of turbo codes raised a lot of attention in the research community and Spielman *et al.* [SS96] and MacKay *et al.* [MN95] rediscovered LDPC codes. LDPC codes were invented by Gallager [Gal62], [Gal63] in the early sixties. However, soon afterwards they were largely forgotten because the excellent error correcting performance of these codes could not be demonstrated on early computers with very limited processing power. LDPC codes are essentially block codes, but can also be interpreted as a serial concatenation of very simple block codes.

In this work we follow the common practice to use the bit error rate (BER) for the performance evaluation of channel codes.

### 2.3.1 Simple Block Codes

We assume that a continuously running sequence of information bits is divided into blocks of $K$ information bits

$$\boldsymbol{u} = (u_0, u_1, \ldots, u_k, \ldots, u_{K-1}) \tag{2.9}$$

and mapped (encoded) to code words

$$\boldsymbol{c} = (c_0, c_1, \ldots, c_i, \ldots, c_{N-1}), \tag{2.10}$$

where $N$ denotes the block length of the code. We solely restrict ourselves to binary codes with $u_k \in \mathbb{F}_2$ and $c_i \in \mathbb{F}_2$. A binary $(N, K)$ block code $\mathcal{C}$ can then be defined as the set of $2^K$ binary code words $\boldsymbol{c}$, $\boldsymbol{c} \in \mathcal{C}$. A code is linear when the sum of two code words again yields a code word. The set of all possible binary $N$-tuples $\boldsymbol{x}$ forms the symbol configuration space $\mathcal{X}$ with $\mathcal{C} \subseteq \mathcal{X}$. The Hamming distance $d_H(\boldsymbol{c}, \boldsymbol{x})$ is then determined by the number of different bit positions between $N$-tuples $\boldsymbol{c}$ and $\boldsymbol{x}$. An important parameter for the characterization of block codes is the minimum distance $d_{min}$. The minimum distance is defined as the minimum Hamming distance between two arbitrary code words $\boldsymbol{c}$ and $\boldsymbol{c}'$. It therefore mainly determines the error correction capability of the code. Further information about the error correcting performance of a code can be gathered from the distribution of the Hamming distance over the set of all code words and, in particular, from the number of code word pairs with minimum distance. The receiver can only guarantee to detect the correct code word when there are less than $d_{min}/2$ transmission errors.

The encoder of block code $\mathcal{C}$ performs a linear and time-invariant bijective (one-to-one) mapping from the set of $2^K$ information vectors $\boldsymbol{u}$ to the set of code words $\boldsymbol{c}$

$$\boldsymbol{u} \in (\mathbb{F}_2)^K \rightarrow \boldsymbol{c} \in (\mathbb{F}_2)^N, \tag{2.11}$$

where $(\mathbb{F}_2)^K$ and $(\mathbb{F}_2)^N$ denote the $K$- and $N$-dimensional vector space over the binary field $\mathbb{F}_2$. The code rate $R$ of the code (or the encoder) is defined as

$$R = \frac{K}{N} \tag{2.12}$$

and represents the fraction of information bits carried in the sequence of code bits. The remaining fraction $1 - R$ represents the amount of redundancy which is available for error detection and error correction in the receiver. The linear encoding rule can be expressed in vector matrix notation as

$$\boldsymbol{c} = \boldsymbol{u}\boldsymbol{G}, \tag{2.13}$$

with $\boldsymbol{G}$ as the $K \times N$ generator matrix of the code. Each code word $\boldsymbol{c}$ can therefore be expressed as a linear combination of the linearly independent row vectors in $\boldsymbol{G}$, which form a basis of the code. A given code $\mathcal{C}$ can be encoded using different generator matrices, i.e., different encoders. We speak about identical codes when two codes share the same set of code words. Different generator matrices for an identical code are referred to as equivalent generator matrices. Two codes $\mathcal{C}$ and $\mathcal{C}'$ are considered to be equivalent when the bit positions in code word $\boldsymbol{c} \in \mathcal{C}$ are simply a rearrangement of the bit positions in code word $\boldsymbol{c}' \in \mathcal{C}'$. One can always find a systematic generator matrix $\boldsymbol{G}_{sys}$ in the form

$$\boldsymbol{G}_{sys} = [\boldsymbol{I}_K | \boldsymbol{P}], \tag{2.14}$$

where $\boldsymbol{I}_K$ represents the $K \times K$ identity matrix and $\boldsymbol{P}$ refers to a $K \times (N - K)$ matrix. The systematic generator matrix in (2.14) guarantees that the $K$ information bits appear unchanged among the first $K$ bit positions in the code word.

For every linear block code $\mathcal{C}$ encoded with generator matrix $\boldsymbol{G}$ there exists an associated $(N-K) \times N$ matrix $\boldsymbol{H}$ with full rank where the row vectors are orthogonal to every code word $\boldsymbol{c}$. The code $\mathcal{C}$ can therefore also be characterized by

$$\boldsymbol{H}\boldsymbol{c}^T = \boldsymbol{0}, \tag{2.15}$$

where $\boldsymbol{H}$ acts as parity-check matrix of the code . Each row of $\boldsymbol{H}$ imposes a parity-check on code word $\boldsymbol{c}$ and a valid code word satisfies all $N - K$ parity-checks. In case of a systematic generator matrix as in (2.14) we can easily obtain a parity-check matrix in the form

$$\boldsymbol{H} = (\boldsymbol{P}^T | \boldsymbol{I}_{N-K}), \tag{2.16}$$

with $\boldsymbol{I}_{N-K}$ as $(N - K) \times (N - K)$ identity matrix.

A simple example of a linear block code is the $(N, K, d_{min}) = (3, 2, 2)$ code $\mathcal{C}$ with $\mathcal{C} = \{000, 011, 101, 110\}$ and rate $R = 2/3$. This code can for example be encoded using the systematic generator matrix

$$\boldsymbol{G} = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix}, \tag{2.17}$$

so that the first $K = 2$ code bits represent the information bits and the last code bit is obtained from a parity-check performed on the two information bits. This code is therefore also referred to as single parity-check (SPC) code. With (2.16) we obtain for the corresponding parity-check matrix of the code

$$\boldsymbol{H} = \begin{pmatrix} 1 & 1 & 1 \end{pmatrix}. \tag{2.18}$$

When this parity-check matrix is used as generator matrix we obtain the associated dual code $\mathcal{C}^\perp$ with $\mathcal{C}^\perp = \{000, 111\}$, i.e., the (3,1,3) repetition code with rate $R^\perp = 1/3$. The codes $\mathcal{C}$ and

$\mathcal{C}^{\perp}$ of this example can be extended to the family of $(N, N-1, 2)$ SPC codes and $(N, 1, N)$ repetition codes, respectively.

Other examples of simple linear block codes are the (7,4,3) Hamming code and the (8,4,4) extended Hamming code. The systematic generator matrix of the (7,4,3) Hamming code is given as

$$\boldsymbol{G} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix}, \tag{2.19}$$

with the corresponding parity-check matrix according to (2.16)

$$\boldsymbol{H} = \begin{pmatrix} 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{pmatrix}. \tag{2.20}$$

The systematic generator matrix of the (8,4,4) extended Hamming code expands the generator matrix of the (7,4,3) Hamming code in (2.19) by one additional column, i.e.,

$$\boldsymbol{G} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \end{pmatrix}. \tag{2.21}$$

The additional code bit increases the minimum distance of the code from three to four. The corresponding parity-check matrix can again be obtained from (2.16)

$$\boldsymbol{H} = \begin{pmatrix} 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}. \tag{2.22}$$

Note that the (8,4,4) extended Hamming code is self-dual, i.e., $\mathcal{C} = \mathcal{C}^{\perp}$.

### 2.3.2 Low-Density Parity-Check (LDPC) Codes

A particularly important class of binary linear block codes are LDPC codes [Gal62], [Gal63]. LDPC codes are defined by a $(N-K) \times N$ parity-check matrix $\boldsymbol{H}$ according to (2.15). It is characteristic to LDPC codes that the total number of ones in $\boldsymbol{H}$ is small with respect to the overall size of the matrix, i.e., the density of the ones in the matrix is particularly low. In the following we refer to the rows and columns of $\boldsymbol{H}$ with $j$ and $i$, respectively. We assume that there are $d_{c,j}$ ones in row $j$ and $d_{v,i}$ ones in column $i$. Row $j$ and column $i$ of the parity-check matrix are illustrated in Fig. 2.5 a) for an example with $d_{c,j} = 3$ and $d_{v,i} = 3$. We speak about regular LDPC codes when there is an equal number of ones in each row and each column, i.e., $d_{c,j} = d_c$, $\forall j$, and $d_{v,i} = d_v$, $\forall i$. Otherwise the codes are considered to be irregular. LDPC codes can be both systematic as well as non-systematic depending on whether the information bits are transmitted or not.

LDPC codes are conveniently visualized by a graphical model as shown in Fig. 2.5 b). Here, code bits and parity-checks are represented as circles. The node of a code bit is connected to the node of a parity-check whenever it participates in the corresponding parity-check equation. For our example in Fig. 2.5 a) with $d_{c,j} = 3$ and $d_{v,i} = 3$ this implies that the $j$-th parity-check node is connected to the nodes representing the first, the $i$-th and the last code bit. On the other hand, the node for the $i$-th code bit participates in the $j$-th parity-check node as well as the first and the last parity-check node.

The representation of codes with code graphs is further investigated in Chapter 3.

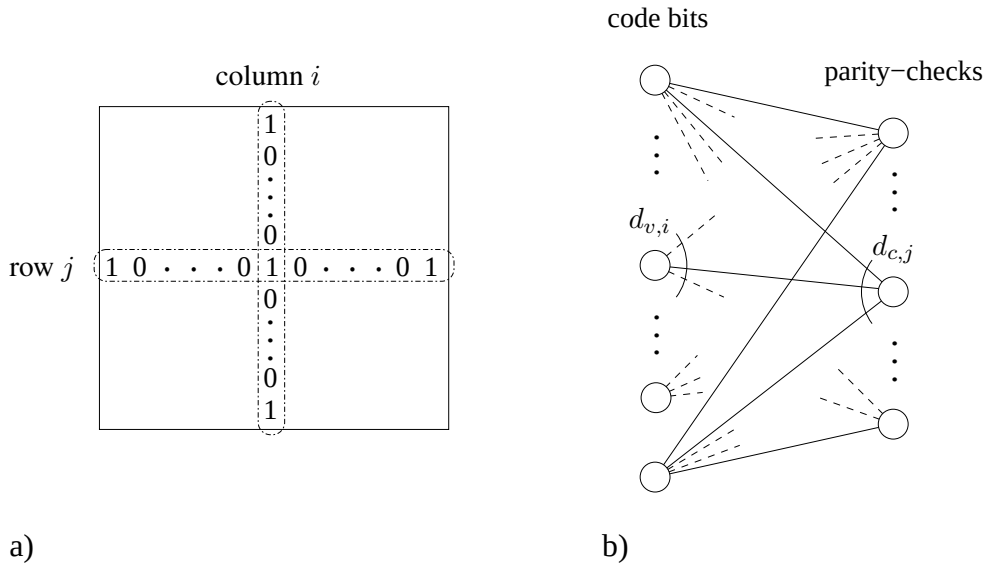



**Figure 2.5**: Example of a row and a column in the parity-check matrix of a LDPC code with $d_{c,j} = 3$ and $d_{v,i} = 3$ in a) and the corresponding graph representation of the code in b).

### 2.3.3 Convolutional Codes

A convolutional encoder operates on a continuously running sequence of information bits

$$\boldsymbol{u} = (\boldsymbol{u}_0, \boldsymbol{u}_1, \ldots, \boldsymbol{u}_k, \ldots) = \left(u_0^{(1)}, \ldots, u_0^{(k_0)}, u_1^{(1)}, \ldots, u_1^{(k_0)}, \ldots, u_k^{(1)}, \ldots, u_k^{(k_0)}, \ldots\right) \quad (2.23)$$

and generates a continuously running sequence of code bits

$$\boldsymbol{c} = (\boldsymbol{c}_0, \boldsymbol{c}_1, \ldots, \boldsymbol{c}_k, \ldots) = \left(c_0^{(1)}, \ldots, c_0^{(n_0)}, c_1^{(1)}, \ldots, c_1^{(n_0)}, \ldots, c_k^{(1)}, \ldots, c_k^{(n_0)}, \ldots\right). \quad (2.24)$$

This is in clear contrast to block codes where one block of information bits is encoded at a time. The encoding of convolutional codes is a linear operation which can be described by

$$\boldsymbol{c}_k = \boldsymbol{u}_k\boldsymbol{G}_0 + \boldsymbol{u}_{k-1}\boldsymbol{G}_1 + \ldots + \boldsymbol{u}_{k-m}\boldsymbol{G}_m, \quad (2.25)$$

with the $k_0 \times n_0$ matrices $\boldsymbol{G}_l$, $l \in \{0, \ldots, m\}$. The semi-infinite code sequence $\boldsymbol{c}$ can then be generated according to

$$\boldsymbol{c} = \boldsymbol{u}\boldsymbol{G}, \quad (2.26)$$

with

$$\boldsymbol{G} = \begin{pmatrix} \boldsymbol{G}_0 & \boldsymbol{G}_1 & \boldsymbol{G}_2 & \cdots & \boldsymbol{G}_m & & \\ & \boldsymbol{G}_0 & \boldsymbol{G}_1 & \boldsymbol{G}_2 & \cdots & \boldsymbol{G}_m & \\ & & \boldsymbol{G}_0 & \boldsymbol{G}_1 & \boldsymbol{G}_2 & \cdots & \boldsymbol{G}_m \\ & & & \ddots & \ddots & \ddots & \end{pmatrix} \quad (2.27)$$

as semi-infinite generator matrix of the code. Note that whenever there are empty areas as in (2.27) the matrix is assumed to be filled with zeros. Equivalent to block codes a convolutional code can also be generated using various different encoders, i.e., generator matrices. The only difference is how the sequence of information bits is mapped onto the sequence of code bits. One sub-matrix of the generator matrix in (2.27) is determined by the binary $k_0 \times n_0$ matrix

$$\boldsymbol{G}_l = \begin{pmatrix} g_{1,l}^{(1)} & g_{1,l}^{(2)} & \cdots & g_{1,l}^{(n_0)} \\ g_{2,l}^{(1)} & g_{2,l}^{(2)} & \cdots & g_{2,l}^{(n_0)} \\ \vdots & \vdots & & \vdots \\ g_{k_0,l}^{(1)} & g_{k_0,l}^{(2)} & \cdots & g_{k_0,l}^{(n_0)} \end{pmatrix}. \quad (2.28)$$

In many cases it is convenient to express the generator matrix in terms of the delay operator $D$ [JZ99]. This allows the representation of the generator matrix in (2.27) with the $k_0 \times n_0$ polynomial generator matrix

$$\boldsymbol{G}(D) = \begin{pmatrix} g_1^{(1)}(D) & g_1^{(2)}(D) & \cdots & g_1^{(n_0)}(D) \\ g_2^{(1)}(D) & g_2^{(2)}(D) & \cdots & g_2^{(n_0)}(D) \\ \vdots & \vdots & & \vdots \\ g_{k_0}^{(1)}(D) & g_{k_0}^{(2)}(D) & \cdots & g_{k_0}^{(n_0)}(D) \end{pmatrix}. \tag{2.29}$$

The entries in row $\eta$, $\eta \in \{1, \ldots, k_0\}$, and column $\xi$, $\xi \in \{1, \ldots, n_0\}$, describe the impact of the $\eta$-th information bits within the $k_0$-tuples of information bits on the $\xi$-th code bits within the $n_0$-tuples of code bits. The code rate of the convolutional code (or encoder) is then defined as

$$R = \frac{k_0}{n_0}. \tag{2.30}$$

The constraint length for the $\eta$-th input of the convolutional encoder is determined by the maximum degree of the polynomials $g_\eta^\xi(D)$, i.e.,

$$\nu_\eta = \max_{1 \le \xi \le n_0} \{\deg(g_\eta^\xi(D))\}. \tag{2.31}$$

The maximum constraint length is then referred to as the encoder memory

$$m = \max_{1 \le \eta \le k_0} \{\nu_\eta\}, \tag{2.32}$$

while the overall constraint length is the sum of all constraint lengths

$$\nu = \sum_{\eta=1}^{k_0} \nu_\eta. \tag{2.33}$$

The Hamming distance and the minimum Hamming distance of convolutional codes are defined analogous to block codes.

We now focus on the realization of a single entry in the polynomial generator matrix in (2.29) so that we can skip the row and column index. In general, each entry in (2.29) represents a rational transfer function

$$g(D) = \frac{f(D)}{q(D)} = \frac{f_0 + f_1 D + \ldots + f_m D^m}{1 + q_1 D + \ldots + q_m D^m}, \tag{2.34}$$

with $f(D)$ and $q(D)$ as the polynomials for the feedforward and the feedback path of the encoder, respectively. The rational transfer function in (2.34) can be realized in controller canonical form as shown in Fig. 2.6. Note that this encoder solely describes the impact of information bits at a particular bit position within the $k_0$-tuple on the code bits at a particular bit position within the $n_0$-tuple. In the overall convolutional encoder other information bits within the $k_0$-tuple at the input may also contribute to the calculation of these code bits. Convolutional encoders utilizing the general encoder structure in Fig. 2.6 with $q(D) \neq 1$ are referred to as feedback encoders or recursive encoders. The input to the shift register is then not only determined by the encoder input $u_k$ but also by the content of the shift register. For the special case when there is no feedback in the encoder, i.e., $q(D) = 1$, we obtain a feedforward encoder where the content of the shift register simply represents the delayed inputs $u_{k-1}, \ldots, u_{k-m}$. Examples of a feedforward and a feedback encoder for a convolutional code with memory $m = 2$

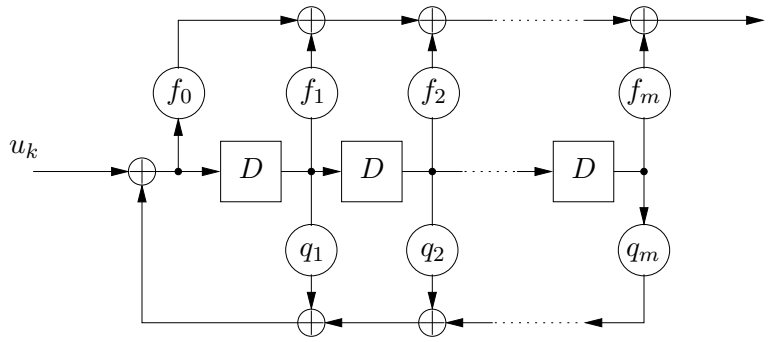



**Figure 2.6**: Realization of the rational transfer function $g(D)$ in controller canonical form.

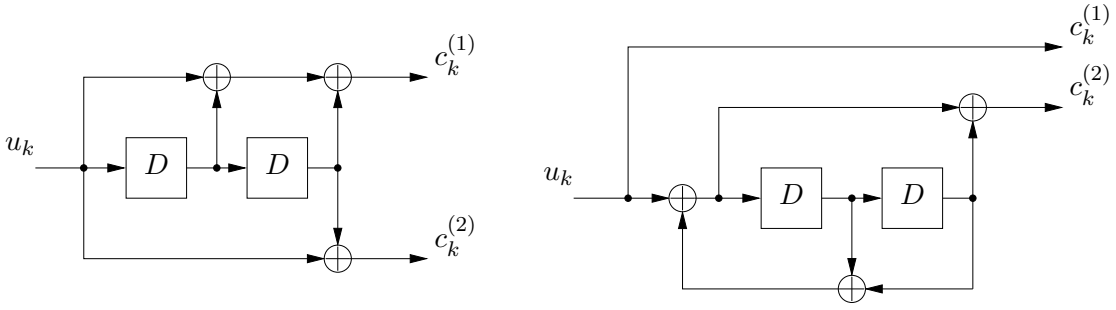



**Figure 2.7**: Encoder realizations for a rate $R = 1/2$ and memory $m = 2$ convolutional code (feedforward encoder in a) and recursive systematic encoder in b) ).

and rate $R = 1/2$ are shown in Fig. 2.7. The feedforward encoder in Fig. 2.7 a) is based on the polynomial generator matrix

$$\boldsymbol{G}(D) = \left(1 + D + D^2 \quad 1 + D^2\right), \tag{2.35}$$

while the feedback encoder in Fig. 2.7 b) realizes

$$\boldsymbol{G}_{sys}(D) = \frac{1}{1 + D + D^2}\,\boldsymbol{G}(D) = \left(1 \quad \frac{1 + D^2}{1 + D + D^2}\right). \tag{2.36}$$

Note that both encoders generate the same code, i.e., the same set of code words. The only difference is the way the sequence of information bits is mapped onto the sequence of code bits. The recursive encoder in Fig. 2.7 b) is also called a systematic encoder since the information bits appear unchanged among the code bits. Such a systematic encoder with the associated systematic generator matrix exists for every convolutional code [JZ99]. The systematic generator matrix in (2.36) can simply be obtained from the polynomial generator matrix in (2.35) through the multiplication with $1/(1 + D + D^2)$.

The encoding process of a convolutional encoder can also be described using the state space representation of linear systems. A state space representation is characterized by the equations

$$\boldsymbol{s}_{k+1} = \boldsymbol{s}_k\boldsymbol{A} + \boldsymbol{u}_k\boldsymbol{B}, \tag{2.37}$$

$$\boldsymbol{c}_k = \boldsymbol{s}_k\boldsymbol{C} + \boldsymbol{u}_k\boldsymbol{D}, \tag{2.38}$$

where $\boldsymbol{A}$ is the $(m \times m)$ state matrix, $\boldsymbol{B}$ the $(k_0 \times m)$ input (control) matrix, $\boldsymbol{C}$ the $(m \times n_0)$ output (observation) matrix and $\boldsymbol{D}$ is the $(k_0 \times n_0)$ transition matrix. The state vector

$$\boldsymbol{s}_k = \left(s_k^{(1)}, \ldots, s_k^{(m)}\right) \tag{2.39}$$

represents internal encoder states, i.e., the content of the $m$ internal memory elements, at time $k$. A convolutional encoder has thus a total number of $2^m$ possible states. The encoder output $\boldsymbol{c}_k$ is determined by the internal encoder state $\boldsymbol{s}_k$ and the encoder input $\boldsymbol{u}_k$. Typically, the encoding process starts in the all-zero state with $\boldsymbol{s}_0 = \boldsymbol{0}$, i.e., all internal memory elements are initialized with zeros. When the output $\boldsymbol{c}_k$ is calculated the encoder changes from state $\boldsymbol{s}_k$ to state $\boldsymbol{s}_{k+1}$. Again, this transition depends on the internal encoder state $\boldsymbol{s}_k$ and the encoder input $\boldsymbol{u}_k$. The state space representation of convolutional codes directly leads to a graphical representation of the code in the form of a code trellis. This trellis representation plays a fundamental role when it comes to decoding of convolutional codes. More details about this can be found in Section 4.4.2.

There are also punctured convolutional codes. Here, particular code bits are removed from the code sequence so that the effective code rate increases accordingly. This for example allows the adjustment of the code rate dependent on the quality of the communication channel while using only a single encoder and decoder. A special case of punctured codes which are of particular practical importance are rate-compatible punctured convolutional (RCPC) codes [Hag88].

### 2.3.4 Tailbiting Convolutional Codes

Convolutional codes are defined for semi-infinite sequences of information bits. However, almost all practical applications require that the information bits are divided into separate blocks of finite size which can be processed independent of each other. A very simple and straightforward method is to stop the encoding process after $K$ information bits. This is called direct truncation of the convolutional code. The main disadvantage of this technique is that the error protection towards the end of the code block becomes inferior. It is therefore more common to terminate the convolutional code by inserting dummy bits into the encoder after a block of $K$ information bits has been encoded. These dummy bits control the encoder in a way that it reaches a pre-defined state, typically the all-zero state. Clearly, this termination of the convolutional code comes at the expense of a rate loss. The termination of the feedforward encoder in Fig. 2.7 a) can simply be achieved by adding $m$ zeros to the $K$ information bits. In case of the feedback encoder in Fig. 2.7 b) the termination works differently. Here, the termination bits depend on the encoder state after encoding the $K$ information bits. This state is referred to in the following as $\boldsymbol{s}_{K_0}$. There are a total number of $2^m$ possible $m$-tuples for $\boldsymbol{s}_{K_0}$ which require dedicated termination bits. It is sufficient to identify these termination bits once and to store them together with the possible ending states in a look-up table. The main advantages of tailbiting codes [SvT79], [MW86] are that there are no such termination bits required (and therefore no rate loss occurs) and that all code bits are protected equally (which is not the case for direct truncation). This makes tailbiting codes particularly interesting for applications which demand codes with short block lengths. Tailbiting codes require that the state of the encoder at the beginning and the end of the encoding process is identical, i.e., $\boldsymbol{s}_0 = \boldsymbol{s}_{K_0}$. The encoder may start in any state as long as it is guaranteed that it also ends in this state. This tailbiting condition is easily fulfilled for the feedforward encoder in Fig. 2.7 a). Here, the convolutional encoder is simply initialized at the beginning of the encoding process with the last $m$ bits of the block of $K$ information bits. These bits are also stored in the encoder after encoding. For the feedback encoder in Fig. 2.7 b) it is more difficult to find the appropriate starting state. This is because in this case the starting state $\boldsymbol{s}_0$ depends on the entire block of information bits. A solution for such recursive convolutional encoders can be found in [WBRC98], [Wei02]. This solution is based on the superposition of the zero-input solution $\boldsymbol{s}_k^{[zi]}$ and the zero-state solution $\boldsymbol{s}_k^{[zs]}$ with $\boldsymbol{s}_k = \boldsymbol{s}_k^{[zi]} + \boldsymbol{s}_k^{[zs]}$. The starting state of the encoder is then determined by

$$\boldsymbol{s}_0 \left( \boldsymbol{A}^{K_0} + \boldsymbol{I}_m \right) = \boldsymbol{s}_{K_0}^{[zs]}, \tag{2.40}$$

with $\boldsymbol{I}_m$ as the $m \times m$ identity matrix. It becomes apparent in (2.40) that the use of a feedback encoder places restrictions on the block length of the tailbiting code. This is because $(\boldsymbol{A}^{K_0}+\boldsymbol{I}_m)$ needs to be invertible and thus $\det(\boldsymbol{A}^{K_0}+\boldsymbol{I}_m) \neq 0$. In the following we always assume that the block length is chosen appropriately. Encoding of a tailbiting code with a recursive encoder then proceeds as follows. In a first step the encoder starts with the all-zero state $\boldsymbol{s}_0 = \boldsymbol{0}$ and determines the zero-state response $\boldsymbol{s}_{K_0}^{[zs]}$ based on the given block of information bits. The output of the encoder is ignored. In the second step the encoder starts with the appropriate starting state $\boldsymbol{s}_0$ which satisfies (2.40) and encodes the tailbiting convolutional code. The appropriate starting states of the encoder can be pre-computed according to (2.40) based on the given block length of the tailbiting code and the $2^m$ possible realizations of the zero-state response $\boldsymbol{s}_{K_0}^{[zs]}$. The starting states can be stored in a look-up table together with the corresponding zero-state response.

More details on tailbiting convolutional codes can be found in, e.g., [Wei02], [JZ99].

### 2.3.5 Parallel Concatenated Convolutional Codes

In 1993 Berrou *et al.* [BGT93] demonstrated that the parallel concatenation of recursive systematic convolutional codes in combination with an iterative processing of the component codes in the receiver achieves an unprecedented error correcting performance close to the theoretical limits. Such concatenations of codes were therefore termed turbo codes. The encoder for the example of the parallel concatenation of two rate $R = 1/2$ convolutional codes with recursive systematic encoder is depicted in Fig. 2.8. There is a separate encoder for component code $\mathcal{C}_1$

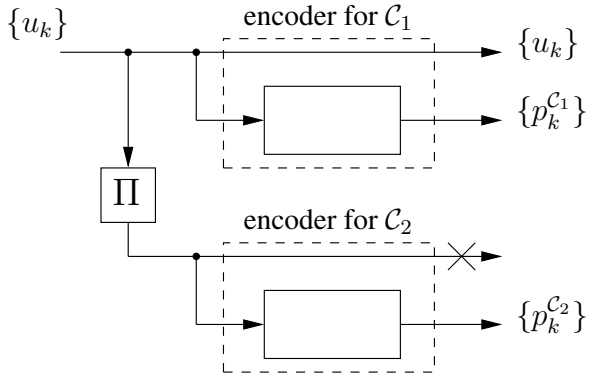



**Figure 2.8**: Encoder for the parallel concatenation of two rate $R = 1/2$ convolutional codes with recursive systematic encoder.

and component code $\mathcal{C}_2$. Both encoders process the same sequence of information bits $\{u_k\}$. The encoder for $\mathcal{C}_1$ uses the given bit order of the information bits while the encoder for $\mathcal{C}_2$ works with a permuted version of these bits. This permutation is achieved with the interleaver $\Pi$. The turbo code then comprises the sequence of information bits $\{u_k\}$ and the output of the two recursive systematic encoders $\{p_k^{\mathcal{C}_1}\}$ and $\{p_k^{\mathcal{C}_2}\}$ as the parity bits with $\boldsymbol{c}_k = \left(u_k, p_k^{\mathcal{C}_1}, p_k^{\mathcal{C}_2}\right)$. Note that the systematic output of the component encoders only needs to be transmitted once. The code rate of the turbo code is then determined by the code rates $R_{\mathcal{C}_1}$ and $R_{\mathcal{C}_2}$ of the component codes according to

$$R = \frac{1}{\frac{1}{R_{\mathcal{C}_1}} + \frac{1}{R_{\mathcal{C}_2}} - 1}. \tag{2.41}$$

Turbo codes were first standardized for a commercial application in the third generation mobile communication system UMTS [ETS00]. Here, the two component codes $\mathcal{C}_1$ and $\mathcal{C}_2$ are encoded with recursive systematic convolutional encoders with rate $R = 1/2$ and memory

$m = 3$. The component encoders are characterized by the polynomial generator matrix

$$\boldsymbol{G}_{UMTS}(D) = \left(1 \quad \frac{1 + D + D^3}{1 + D^2 + D^3}\right). \tag{2.42}$$

The block length of this turbo scheme is in the range $40 \leq K \leq 5114$ information bits. Both component codes are terminated so that the overall code rate is $R = K/N = K/(3K+12)$ with $R$ ranging from 0.3030 to 0.3331. Depending on the block length of the turbo code different interleavers are specified [ETS00].

Turbo codes are also standardized for the DVB interaction channel for satellite distributed systems [ETS03]. Here, the parallel concatenation of two recursive systematic tailbiting convolutional codes with memory $m = 3$ is utilized. The component encoders are described by the polynomial generator matrix

$$\boldsymbol{G}_{DVB}(D) = \left(1 \quad 1 \quad \frac{1 + D^2 + D^3}{1 + D + D^3} \quad \frac{1 + D^3}{1 + D + D^3}\right). \tag{2.43}$$

This standard specifies different block lengths as well as different code rates which are obtained through puncturing of the component codes. A rate $R = 1/2$ turbo code for example requires two rate 2/3 component codes, compare (2.41). In this case the second parity bit obtained from the generator matrix in (2.43) is punctured.

The above turbo codes for UMTS and DVB are further investigated in Chapter 5.

## 2.4 A primer on analog FEC decoders

The task of a FEC decoder is to reconstruct the vector of information bits $\boldsymbol{u}$ based on the vector of noisy received samples $\boldsymbol{y}$. It exploits the redundancy present in the transmitted code word $\boldsymbol{c}$ in order to correct potential transmission errors. FEC decoders are commonly implemented on signal processors operating in the digital domain. In this work we deal with a new approach where FEC decoders are realized by means of analog signal processors. The use of analog signal processing promises higher operating speed, lower power consumption and/or smaller area compared to a conventional digital realization. Analog decoders rely on the time- and value-continuous operation of analog transistor circuits which is in clear contrast to the plain switching operation between zero and one of digital circuits. A digital FEC decoder requires that the L-values $L_c y_i$ received from the communication channel, i.e., the matched filter output $y_i$ weighted with the channel state information $L_c$, are converted into a digital signal representation. For this, the L-values need to be represented by a sufficiently large number of bits in order to preserve an adequate amount of reliability information about the received samples. Analog signal processing in the FEC decoder naturally captures the complete information contained in the received samples so that there is no quantization loss. Furthermore, reliability information is preserved throughout the decoder. The computational cores of our analog decoding networks are based on two elementary transistor circuits as shown in Fig. 2.9 [MGYH00], [MHO00]. It is characteristic to our circuit design that L-values are represented in terms of differential voltages. When we apply such a differential voltage $\Delta V$ to the input of a differential transistor pair as shown in Fig. 2.9 a) it determines how the bias current $I$ is split up into the currents $I_0$ and $I_1$. We will demonstrate later in more detail that the two transistors effectively transform the L-value (interpreted as differential voltage $\Delta V$) into the associated probabilities $P(c_i = 0)$ and $P(c_i = 1)$ represented by $I_0/I$ and $I_1/I$, respectively. The pair of diode-connected transistors in Fig. 2.9 b) performs the inverse operation of the circuit in Fig. 2.9 a). It transforms the two input currents $I_0$ and $I_1$ into the differential output voltage $\Delta V$. When we treat the two input currents as probabilities as in the above we find that $\Delta V$ again represents the associated L-value. Generalizations of the two elementary transistor circuits in Fig. 2.9 are sufficient in

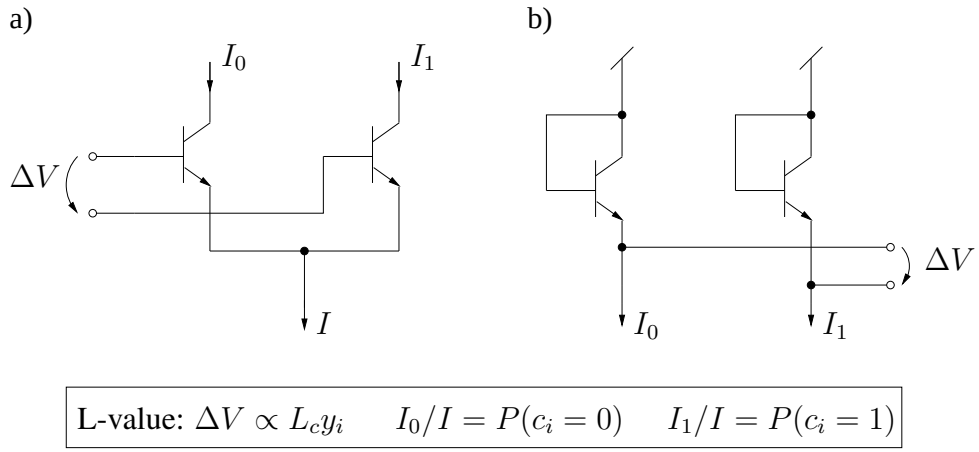



**Figure 2.9**: Conversion of a L-value into the corresponding probabilities in a) and the conversion of the probabilities back to a L-value in b).

order to construct arbitrary computational cores for our analog decoders. It will unfold in the course of this thesis that very simple configurations of these analog circuits implement the ideal decoder operations while digital decoder realizations need to rely on approximations in order to reduce complexity, see, e.g., [RVH95], [Daw96]. Analog decoders may thus achieve optimal decoder performance in terms of the BER since quantization loss and the approximation of exact computations is avoided.

## 2.5 Mutual Information and Channel Capacity

The mutual information of two continuous random variables $X$ and $Y$ with joint pdf $p(x, y)$ and marginal pdfs $p(x)$ and $p(y)$ is given as

$$I(X;Y) = \int_{-\infty}^{+\infty} \int_{-\infty}^{+\infty} p(x)p(y|x) \log_2 \frac{p(y|x)p(x)}{p(x)p(y)} \, dx \, dy. \tag{2.44}$$

It is a measure for the information available about $X$ in case $Y$ has been observed. The mutual information ranges between zero and one and the unit is bit. For statistically independent $X$ and $Y$ we have $I(X;Y) = 0$ while for $I(X;Y) = 1$ the variable $X$ is completely described by the observation $Y$. In case of a time-discrete communication channel with binary input $x \in \{+1, -1\}$ and analog output $y$ we obtain

$$I(X;Y) = \sum_{x \in \{+1,-1\}} \int_{-\infty}^{+\infty} P(x)p(y|x) \log_2 \frac{p(y|x)}{p(y)} \, dy. \tag{2.45}$$

The mutual information in (2.45) is maximized for equally likely input bits with $P(x = +1) = P(x = -1) = 0.5$, thus

$$I(X;Y) = \frac{1}{2} \sum_{x \in \{+1,-1\}} \int_{-\infty}^{+\infty} p(y|x) \log_2 \frac{p(y|x)}{p(y)} \, dy \tag{2.46}$$

$$= \frac{1}{2} \sum_{x \in \{+1,-1\}} \int_{-\infty}^{+\infty} p(y|x) \log_2 \frac{2p(y|x)}{p(y|x = +1) + p(y|x = -1)} \, dy. \tag{2.47}$$

In 1948 Claude E. Shannon defined the capacity of the communication channel as the maximum mutual information [Sha48], i.e.,

$$C = \max_{P(x)}\{I(X;Y)\}, \tag{2.48}$$

where the maximization is carried out over all distributions $P(x)$ at the input. The channel capacity $C$ is fundamental part of Shannon's celebrated *noisy channel coding theorem* [Sha48] where it is stated that there exists a reliable transmission with an arbitrarily small error probability as long as

$$R < C. \tag{2.49}$$

On the other hand, when the code rate $R$ exceeds the channel capacity a reliable transmission is not possible. Shannon demonstrated that a noisy communication channel only limits the transmission rate of the information in bits per channel use and not the quality of the transmission. This theorem was revolutionary since the noise was commonly expected to limit the reliability of a transmission. It was clear that the channel capacity can only be approached when codes with large block lengths are used, i.e., $N \rightarrow \infty$. Unfortunately, no further hint was given for the design of such codes.

The channel capacity $C$ of the AWGN channel is evaluated numerically in Fig. 2.10 and Fig. 2.11 over $E_S/N_0$ in dB and $E_b/N_0$ in dB, respectively.

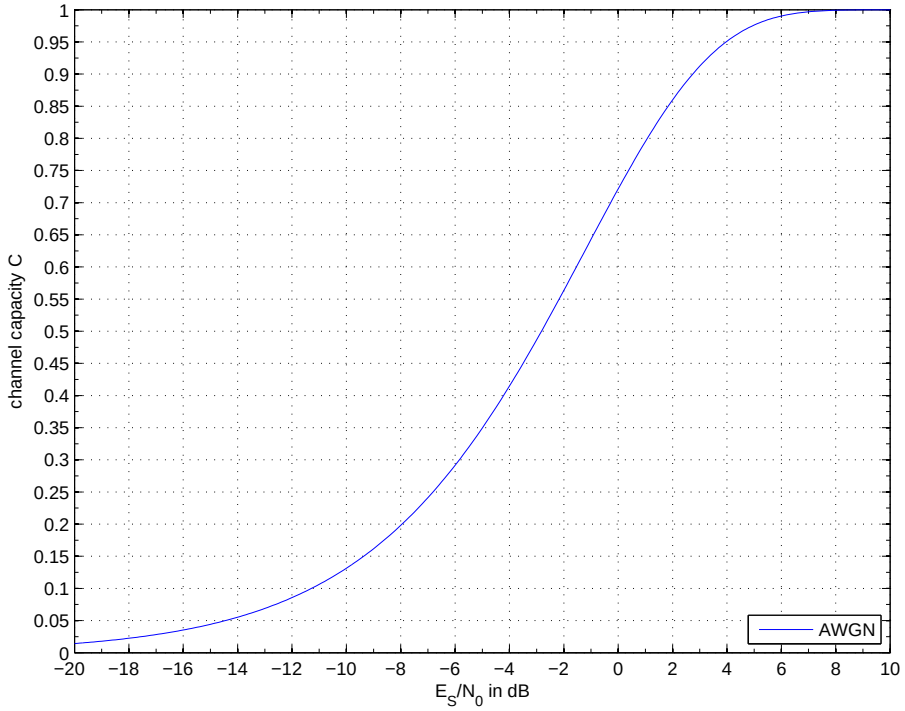



**Figure 2.10**: Channel capacity of the binary input AWGN channel over $E_S/N_0$.

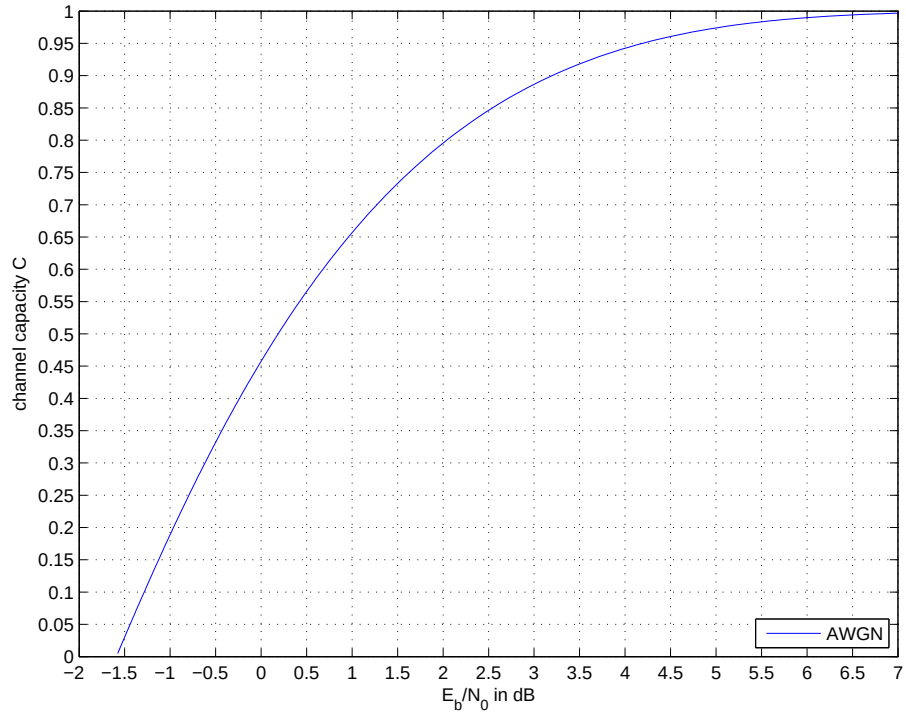



**Figure 2.11**: Channel capacity of the binary input AWGN channel over $E_b/N_0$.

# 3

# *Codes on Graphs*

Codes are conveniently described and visualized by graphical models. These models comprise a set of symbol variables and a set of local constraints which are connected to each other in the code graph. The LDPC codes of Gallager [Gal62], [Gal63] were probably the first codes which utilized a code graph. Here, the symbol variables correspond to code bits and the local constraints are defined through parity-check equations. The work on code graphs was further elaborated by Tanner [Tan81], who also introduced generalized constraint nodes. The introduction of state variables into the Tanner graph by Wiberg *et al.* [Wib96], [WLK95] was a major step in establishing a link to code trellises and turbo codes. This contribution directly led to what is nowadays known as factor graphs [FKLW97], [KFL01]. Later, Forney imposed some degree restrictions on the variables in the factor graph. These restrictions gave rise to a fundamental duality theorem where the same graph topology can be facilitated for a code $\mathcal{C}$ and its dual code $\mathcal{C}^{\perp}$ [For01]. Such graphs are referred to as normal graphs. The intense interest in code graphs is also motivated by the fact that many popular decoding algorithms are based on graphical code descriptions. In the scope of this work normal graphs are particularly important since they can be utilized for the representation of analog decoders in a manner similar to a block diagram. There are many different normal graphs for a given code which exhibit different topologies and may involve loops of different size. This is demonstrated in the following for the example of the (8,4,4) extended Hamming code. Note that, in general, decoders based on different code graphs yield a different decoder performance.

After the definition of some mathematical notation in Section 3.1 we start with Tanner graphs in Section 3.2. We then turn to factor graphs and normal graphs in Section 3.3 and Section 3.4, respectively. Normal graphs are then transformed in a way that each node in the code graph is only connected to three neighboring nodes, i.e., the degree is three. This step is motivated by the circuit implementation of analog decoders as we will see later in Chapter 6.

## 3.1 Mathematical Notation

Let us introduce some mathematical notation [For01] which is useful for the description of codes on graphs. We refer to variables by upper case letters and to the values, i.e., realizations, of the variables by the corresponding lower case letters. The alphabet of a variable is denoted by the corresponding upper case script letter. The variable $A_i$ can then take on values $a_i \in \mathcal{A}_i$ where $\mathcal{A}_i$ can be either a vector space over a finite field, e.g., the set $(\mathbb{F}_2)^n$ of all binary $n$-tuples, or a finite Abelian group. A collection of symbol variables $A_i$ defined by the discrete (and finite)

index set $i \in I_{\mathcal{A}}$ forms the symbol configuration space $\mathcal{A}$ which is obtained by the Cartesian product

$$\mathcal{A} = \prod_{i \in I_{\mathcal{A}}} \mathcal{A}_i. \tag{3.1}$$

Here, the elements $\boldsymbol{a} \in \mathcal{A}$ represent possible symbol configurations.

We can define a code as a subset $\mathcal{C} \subseteq \mathcal{A}$ of the symbol configuration space where the elements $\boldsymbol{c} \in \mathcal{C}$ represent valid symbol configurations, i.e., the code words of the code. We talk about a linear code $\mathcal{C}$ when each symbol alphabet $\mathcal{A}_i$ is a vector space over a (usually finite) field $\mathbb{F}$ and about a binary code when the field is $\mathbb{F}_2$, i.e., $\{0, 1\}$. When each symbol alphabet $\mathcal{A}_i$ is a group we obtain a group code $\mathcal{C}$. The check whether a given symbol configuration is a valid code word or not is commonly performed using a set of local constraints, or local codes, $\mathcal{C}_j$ with the associated discrete index set $j \in I_{\mathcal{C}}$, which is not necessarily related to the index set $I_{\mathcal{A}}$ of the symbols. Such a local constraint $\mathcal{C}_j$ involves only a subset of the symbol variables indexed by a subset $I_{\mathcal{A}}(j)$ of the symbol index set $I_{\mathcal{A}}$. The local code is then defined as a subset of the local symbol configuration space

$$\mathcal{C}_j \subseteq \mathcal{A}_j = \prod_{i \in I_{\mathcal{A}}(j)} \mathcal{A}_i. \tag{3.2}$$

The subset $\mathcal{C} \subseteq \mathcal{A}$ can be defined by a set of local constraints introduced by, e.g., a set of parity-check equations according to $\boldsymbol{c}\boldsymbol{H}^T = \boldsymbol{0}$ such that

$$\mathcal{C} = \{\boldsymbol{c}\boldsymbol{H}^T = \boldsymbol{0} | \boldsymbol{c} \in (\mathbb{F}_2)^N\}. \tag{3.3}$$

The local code $\mathcal{C}_j$ is then defined by the $j$-th row of $\boldsymbol{H}$ with $j \in I_{\mathcal{C}}$ as the row index. The positions of the ones in row $j$ define the subset of symbol variables $A_i$, $i \in I_{\mathcal{A}}(j)$ involved in this local constraint. Each local constraint $\mathcal{C}_j$ determines a set of valid local symbol configurations

$$\boldsymbol{a}_{|I_{\mathcal{A}}(j)} = \{a_i, i \in I_{\mathcal{A}}(j)\}, \tag{3.4}$$

where $\boldsymbol{a}_{|I_{\mathcal{A}}(j)}$ denotes the projection of a symbol configuration $\boldsymbol{a}$ onto the symbols $A_i$ indexed by $I_{\mathcal{A}}(j)$. The overall code $\mathcal{C}$ is then defined as the set of all symbol configurations which satisfy all local constraints

$$\mathcal{C} = \{\boldsymbol{a} \in \mathcal{A} | \boldsymbol{a}_{I_{\mathcal{A}}(j)} \in \mathcal{C}_j, \forall j \in I_{\mathcal{C}}\}. \tag{3.5}$$

## 3.2 Tanner Graphs

A $(N, K)$ block code $\mathcal{C}$ can be represented by a so called Tanner graph [Tan81] which is a graphical representation of the $(N - K) \times N$ parity-check matrix $\boldsymbol{H}$ of the code. This implies that the Tanner graph is not unique since there are many different parity-check matrices for a given code. The Tanner graph consists of binary variable nodes $X_i$ with $x_i \in \mathcal{X}_i$, $\mathcal{X}_i = \mathbb{F}_2$, for the code bits and binary check nodes for the parity-check equations of the local codes $\mathcal{C}_j$. Variable nodes and check nodes are indexed with $i \in I_{\mathcal{X}}$, $I_{\mathcal{X}} = \{1, \ldots, N\}$, and $j \in I_{\mathcal{C}}$, $I_{\mathcal{C}} = \{1, \ldots, N - K\}$, respectively. The variable nodes are represented in the code graph as circles while the $\oplus$ symbol is used for check nodes. Variable node $X_i$ is connected to check node $j$ whenever the variable (code bit) participates in the parity-check equation of the local code $\mathcal{C}_j$, i.e., $i \in I_{\mathcal{X}}(j)$. The degree of check node $j$ is given by $d_{c,j} = |I_{\mathcal{X}}(j)|$ and determines the number of variable nodes connected to the $j$-th check node, i.e., the number of ones in the $j$-th row of $\boldsymbol{H}$. Similarly, the degree of variable node $X_i$ is defined as the number of ones in the $i$-th column of $\boldsymbol{H}$.

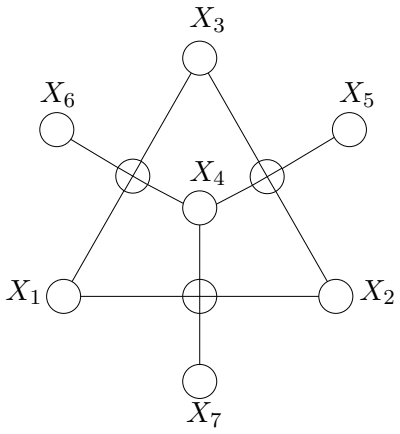



**Figure 3.1**: Tanner graph representation of the (7,4,3) Hamming code.

The Tanner graph representation of the (7,4,3) Hamming code with the parity-check matrix

$$\boldsymbol{H} = \begin{pmatrix} 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{pmatrix}$$

is shown in Fig. 3.1. The code bits $\boldsymbol{c} = (x_1, x_2, x_3, x_4, x_5, x_6, x_7)$ occur in the graph as variable nodes $X_1$ to $X_7$. Here, the first four bits and the last three bits represent the information bits and the parity bits, respectively.

In the following sections we exploit the example of the (8,4,4) extended Hamming code in order to demonstrate that different graph representations of a code are possible. We start with the systematic generator matrix of the (8,4,4) extended Hamming codes as given in (2.21), i.e.

$$\boldsymbol{G} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \end{pmatrix}. \tag{3.6}$$

We now derive a minimum span generator matrix (MSGM) from (3.6) using a simple algorithm based on row operations [McE96]. For this, we define the span of a row vector as the index of the rightmost nonzero entry minus the index of the leftmost nonzero entry. We then obtain the MSGM for code $\mathcal{C}$ by minimizing the overall span of all rows in the generator matrix. It is apparent that the systematic generator matrix in (3.6) with a total span of 21 is clearly not a MSGM. Applying the algorithm from [McE96] we obtain the MSGM for this code with a total span of 16. Such a MSGM can be found for any linear block code. The total span can be further reduced when we allow the permutation of the 4-th and the 8-th column in (3.6). This leads to a generator matrix of an equivalent code. When we apply the algorithm again we obtain

$$\boldsymbol{G}_{MSGM} = \begin{pmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \end{pmatrix} \tag{3.7}$$

with an overall span of only 14. Since the (8,4,4) extended Hamming code is self-dual this matrix is equivalent to the minimum span parity-check matrix (MSPCM) of the code. The Tanner graph representation of the minimum span matrix in (3.7) is depicted in Fig. 3.2. Note that the

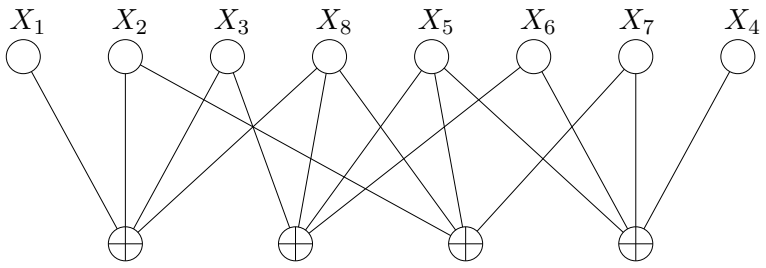



**Figure 3.2**: Tanner graph representation of the (8,4,4) extended Hamming code based on $\boldsymbol{G}_{MSGM}$ with a minimum loop size of four.

minimum girth, i.e., the minimum loop size, of this code graph is four. Minimum span (generator or parity-check) matrices play an important role when it comes to trellis representations with a minimum number of states and branches. Such a minimal trellis representation of the (8,4,4) extended Hamming code is derived in the next section.

## 3.3 Factor Graphs

It is characteristic of Tanner graphs [Tan81] that variable nodes correspond to code symbols which can be observed as part of the code word. Tanner graphs were generalized by Wiberg *et al.* [WLK95], [Wib96] by introducing latent (or hidden) state variables into the code graph. These state variables represent internal states, e.g., internal encoder states, which are not transmitted as part of the code word. A typical example of a code graph which naturally exhibits state variables is the trellis representation of a code. Symbol and state variables are in general non-binary. This necessitates the definition of generalized constraint nodes which were in a similar fashion also introduced by Tanner [Tan81]. These constraint nodes can be seen as generalizations of the LDPC codes of Gallager [Gal62]. More recent work summarized these ideas in a graphical model called factor graph [FKLW97], [KFL01].

We start with Tanner graphs and derive different code graphs including trellis and tailbiting trellis representations. For this, we introduce state variables into the code graph which, in general, lower the degree of constraint nodes, i.e., the number of variable nodes they are connected to. In the context of this thesis we will exploit state variables in order to obtain constraint nodes with a degree of no more than three. We distinguish between code graphs with binary symbol and state variables and code graphs with non-binary variables. First, we present a simple algorithm based on the parity-check matrix $\boldsymbol{H}$ which introduces state variables in a way that the binary graph structure is maintained. We then formulate another algorithm which leads to non-binary variable nodes and generalized constraint nodes. Based on the minimum span parity-check matrix of the (8,4,4) extended Hamming code we then derive the minimal trellis and the minimal tailbiting trellis of the code.

### 3.3.1 State Variables

In the following we extend the notation to the general case of code graphs with state variables [For01]. Here, the code $\mathcal{C} \subseteq \mathcal{X}$ is now defined through a set of local codes $\mathcal{C}_j$, $j \in I_{\mathcal{C}}$ which involve besides the symbol variables $X_i$, $i \in I_{\mathcal{X}}$, also state variables $S_l$, $l \in I_{\mathcal{S}}$. The index sets $I_{\mathcal{C}}$, $I_{\mathcal{X}}$ and $I_{\mathcal{S}}$ are in general unordered and not directly related to each other. Each local code $\mathcal{C}_j$ includes a subset of the symbol and state variables indexed by a subset $I_{\mathcal{X}}(j)$ and $I_{\mathcal{S}}(j)$ of the index sets $I_{\mathcal{X}}$ and $I_{\mathcal{S}}$, respectively. Again, we define the symbol configuration space as the Cartesian product

$$\mathcal{X} = \prod_{i \in I_{\mathcal{X}}} \mathcal{X}_i \tag{3.8}$$

and the state configuration space as the Cartesian product

$$\mathcal{S} = \prod_{l \in I_{\mathcal{S}}} \mathcal{S}_l. \tag{3.9}$$

The set of all global symbol and state configurations satisfying

$$\mathfrak{B} = \{(\boldsymbol{x}, \boldsymbol{s}) \in \mathcal{X} \times \mathcal{S} | \left(\boldsymbol{x}_{|I_{\mathcal{X}}(j)}, \boldsymbol{s}_{|I_{\mathcal{S}}(j)}\right) \in \mathcal{C}_j, j \in I_{\mathcal{C}}\} \tag{3.10}$$

is referred to as the full behavior $\mathfrak{B} \subseteq \mathcal{X} \times \mathcal{S}$ which defines a set of extended code words $\tilde{\boldsymbol{c}} = (\boldsymbol{x}, \boldsymbol{s})$ combining symbol and state variables. The code itself is then the projection $\mathcal{C} = \mathfrak{B}_{|\mathcal{X}}$ onto the symbol variables. This removes all internal state variables so that we obtain the set of original code words again. The local code $\mathcal{C}_j$ is defined as a subset

$$\mathcal{C}_j \subseteq = \left(\prod_{i \in I_{\mathcal{X}}(j)} \mathcal{X}_i\right) \times \left(\prod_{l \in I_{\mathcal{S}}(j)} \mathcal{S}_l\right) \tag{3.11}$$

of the corresponding local Cartesian product configuration space and defines the set of valid local symbol and state configurations

$$\left(\boldsymbol{x}_{|I_{\mathcal{X}}(j)}, \boldsymbol{s}_{|I_{\mathcal{S}}(j)}\right) = \{\{x_i, i \in I_{\mathcal{X}}(j)\}, \{s_l, l \in I_{\mathcal{S}}(j)\}\} \in \mathcal{C}_j. \tag{3.12}$$

In general, the factor graph consists of non-binary variable nodes for symbol and state variables and local check nodes. In order to distinguish between symbol and state variables in the factor graph we represent them with circles and double circles, respectively. Local check nodes are illustrated by either the $\oplus$ symbol or simply by a black box depending on the required operations and the alphabet of the participating variables. Symbol variables $X_i$ and state variables $S_l$ are connected to check node $j$ whenever the variables participate in the local code $\mathcal{C}_j$ with $i \in I_{\mathcal{X}}(j)$ and $l \in I_{\mathcal{S}}(j)$. The degree of a local check node $j$ is given by $|I_{\mathcal{X}}(j)| + |I_{\mathcal{S}}(j)|$ as the number of symbol and state variables connected to it. The degree of a symbol node $i$ and a state node $l$ is defined equivalently as the number of local check nodes connected to it.

For now, we restrict ourselves to binary symbol and state variables, i.e., $\mathcal{X}_i = \mathbb{F}_2$, $i \in I_{\mathcal{X}}$ and $\mathcal{S}_l = \mathbb{F}_2$, $l \in I_{\mathcal{S}}$. In this case, the full behavior $\mathfrak{B}$ can be defined as

$$\mathfrak{B} = \left\{\tilde{\boldsymbol{c}}\tilde{\boldsymbol{H}}^T = \boldsymbol{0} | \tilde{\boldsymbol{c}} \in (\mathbb{F}_2)^{|I_{\mathcal{X}}|+|I_{\mathcal{S}}|}\right\} \tag{3.13}$$

using an $|I_{\mathcal{C}}| \times (|I_{\mathcal{A}}| + |I_{\mathcal{S}}|)$ extended parity-check matrix $\tilde{\boldsymbol{H}}$. We now derive such an extended parity-check matrix for the simple example of a (5,4,2) SPC code. This code is defined by the parity-check matrix

$$\boldsymbol{H} = (11111), \tag{3.14}$$

with $\boldsymbol{c} = (u_1, u_2, u_3, u_4, p)$ and $p = u_1 \oplus u_2 \oplus u_3 \oplus u_4$. The corresponding factor graph is depicted in Fig. 3.3 a). Note that in this case the factor graph is equivalent to the Tanner graph. We now introduce two state variables $S_1$ and $S_2$ whose values are determined by $s_1 = u_1 \oplus u_2$ and $s_2 = u_4 \oplus p$, hence $s_1 \oplus s_2 = u_3$. This expands the original parity-check matrix in (3.14) in both horizontal and vertical direction and we obtain for $\tilde{\boldsymbol{H}}$

| $u_1$ | $u_2$ | $u_3$ | $u_4$ | $p$ | $s_1$ | $s_2$ |
|---|---|---|---|---|---|---|
| 1 | 1 | | | | 1 | |
| | | 1 | | | 1 | 1 |
| | | | 1 | 1 | | 1 |

. (3.15)

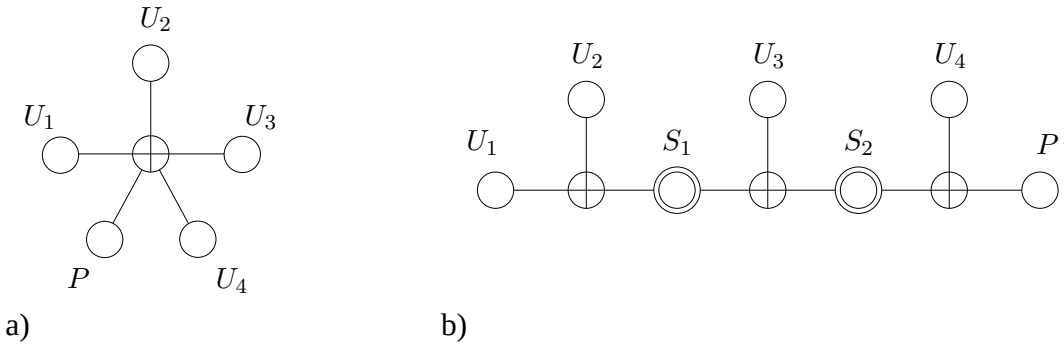



**Figure 3.3**: Two different factor graph representations of the (5,4,2) SPC code.

We use this table format instead of conventional matrix notation and only show the "1"s in the matrix. The row and column weight is here defined as the number of "1"s in the corresponding row and column of the matrix, respectively. We notice that the single parity-check of degree five is now replaced by three local parity-checks of degree three. The associated factor graph is depicted in Fig. 3.3 b). In general, any $(N, N-1, 2)$ SPC code can be split up into $N-2$ check nodes of degree three using $N-3$ internal state variables. We now introduce a simple and straightforward algorithm which generates extended parity-check matrices by introducing new state variables. These state variables are added in a way that the degree of all check nodes in the code graph is limited to three. This algorithm is referred to in the following as the C3 Algorithm.

The C3 Algorithm is divided into three steps:

- STEP 1: Identify all rows in the original matrix with $d_{c,j} > 3$, $j \in I_{\mathcal{C}}$.

- STEP 2: Add $d_{c,j} - 3$ new rows for each row identified in STEP 1. Distribute the $d_{c,j}$ ones in the original row over the resulting $d_{c,j} - 2$ rows in a way that the first (i.e., the original) and the last row has weight two and all other rows have weight one. Maintain the column position of the ones.

- STEP 3: Add $d_{c,j} - 3$ new columns for each row identified in STEP 1. Add two ones in each of these columns so that all the $d_{c,j} - 2$ rows obtained from one row in the original matrix are linked pairwise in sequential order, i.e., the 1st and the 2nd, the 2nd and the 3rd, and so forth. All rows in the matrix then have weight three.

Note that STEP 2 and 3 expand the matrix in vertical and horizontal direction, respectively. Each row of the expanded matrix represents a degree three parity-check and each column added in STEP 3 corresponds to a new state variable. STEP 2 and STEP 3 can be reversed by adding up all rows which originated from one row in the parity-check matrix $\boldsymbol{H}$. This yields the original rows of $\boldsymbol{H}$ and all columns added in STEP 3 have only zero entries, i.e., the corresponding state variables become redundant.

When we apply the C3 Algorithm to the (8,4,4) extended Hamming code with the minimum

span matrix in (3.7) we obtain the $8 \times 12$ matrix $\tilde{\boldsymbol{H}}$

| $x_1$ | $x_2$ | $x_3$ | $x_8$ | $x_5$ | $x_6$ | $x_7$ | $x_4$ | $s_1$ | $s_2$ | $s_3$ | $s_4$ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | | | | | | | 1 | | | |
| | | 1 | 1 | | | | | 1 | | | |
| | | 1 | 1 | | | | | | 1 | | |
| | | | | 1 | 1 | | | | 1 | | |
| | | | | 1 | 1 | | | | | 1 | |
| | | | | | | 1 | 1 | | | 1 | |
| | 1 | | 1 | | | | | | | | 1 |
| | | | | 1 | | 1 | | | | | 1 |

(3.16)

The expanded code vector is given by

$$\tilde{\boldsymbol{c}} = (\boldsymbol{c}, \boldsymbol{s}) = (x_1, x_2, x_3, x_8, x_5, x_6, x_7, x_4, s_1, s_2, s_3, s_4) \tag{3.17}$$

with $\tilde{\boldsymbol{H}}\tilde{\boldsymbol{c}}^T = \boldsymbol{0}$. Here $\boldsymbol{s} = (s_1, s_2, s_3, s_4)$ represent possible realizations of state variables $S_1$ to $S_4$. Note that the position of the 4-th and 8-th code bit are interchanged with respect to the systematic generator matrix in (3.6).

We now extend the C3 Algorithm by two optional steps:

- STEP 4: Perform row operations in order to reduce the row weight to two, if possible. In this case the two "1"s typically occur in the section for state variables.
- STEP 5: A row weight of two obtained in STEP 4 indicates identical state variables. Replace the two state variables with a single state variable, i.e., replace the corresponding columns with the sum of the two. Remove the row with weight two.

STEP 4 of the C3 Algorithm yields the $8 \times 12$ matrix

| $x_1$ | $x_2$ | $x_3$ | $x_8$ | $x_5$ | $x_6$ | $x_7$ | $x_4$ | $s_1$ | $s_2$ | $s_3$ | $s_4$ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | | | | | | | 1 | | | |
| | | 1 | 1 | | | | | 1 | | | |
| | | | | | | | | 1 | 1 | | |
| | | | | 1 | 1 | | | | 1 | | |
| | | | | | | | | | 1 | 1 | |
| | | | | | | 1 | 1 | | | 1 | |
| | 1 | | 1 | | | | | | | | 1 |
| | | | | 1 | | 1 | | | | | 1 |

(3.18)

This matrix is obtained by replacing the third row in (3.16) with the sum of the second and third row, and the fifth row in (3.16) with the sum of the forth and fifth. We find that the third and the fifth row in (3.18) have a row weight of two with $s_1 = s_2$ and $s_2 = s_3$, respectively. When we apply STEP 5 of the above algorithm to the fifth row in (3.18) we obtain the $7 \times 11$ matrix

| $x_1$ | $x_2$ | $x_3$ | $x_8$ | $x_5$ | $x_6$ | $x_7$ | $x_4$ | $s_1$ | $s_2$ | $s_3$ |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | | | | | | | 1 | | |
| | | 1 | 1 | | | | | 1 | | |
| | | | | | | | | 1 | 1 | |
| | | | | 1 | 1 | | | | 1 | |
| | | | | | | 1 | 1 | | 1 | |
| | 1 | | 1 | | | | | | | 1 |
| | | | | 1 | | 1 | | | | 1 |

(3.19)

Note that we omit STEP 5 of the C3 Algorithm for the third row in (3.18) which would lead to a column weight of four. This is because we also restrict the degree of variable nodes in Section 3.4.2. Further note that the total number of "1"s in the matrix reduces from 24 in (3.18) to 20 in (3.19) while the minimum girth increases from four to six. The corresponding factor graph representation of the code is depicted in Fig. 3.4.

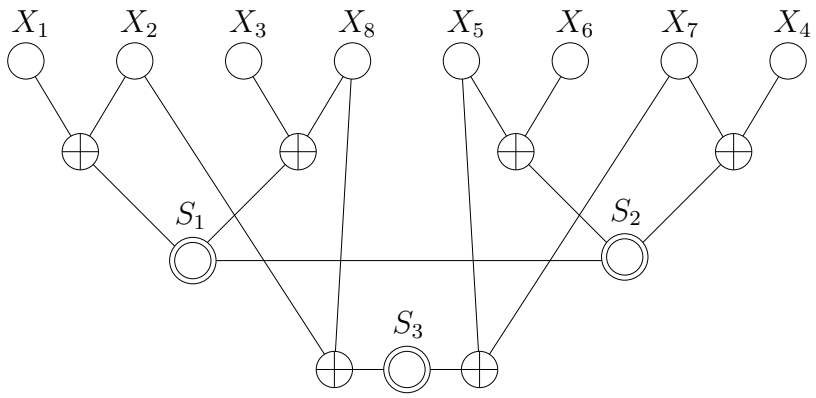



**Figure 3.4**: Factor graph representation of the (8,4,4) extended Hamming code with minimum girth six.

### 3.3.2 Generalized Constraint Nodes

Symbol and state variables in factor graphs are in general non-binary. Such non-binary variables necessitate more generalized constraint nodes as they are, e.g., required for the representation of convolutional codes where $\mathcal{X}_i = (\mathbb{F}_2)^{n_0}$ and $\mathcal{S}_l = (\mathbb{F}_2)^m$. Generalized constraint nodes differ from constraint nodes in Tanner graphs in a way that usually more than one parity-check equation is required in order to fully specify the local code. In the following we derive a factor graph for the trellis representation of the (8,4,4) extended Hamming code. There are different ways to derive a trellis representation of a linear block code using either the generator matrix or the parity-check matrix. Based on a matrix with minimum span we obtain the corresponding minimal trellis representation of the code which minimizes the number of states [McE96], see also [KS95], [KDM$^+$96]. In order to obtain generalized constraint nodes we can use the following simple algorithm:

- STEP 1: Divide the code bits, i.e., the columns of the original matrix, into sub-groups of $n$ bits and associate each group with a symbol group index $i'$. The group span of a row vector is then defined as the index of the rightmost group with a non-zero entry minus the index of the leftmost group with a non-zero entry.

- STEP 2: Starting with the first non-zero symbol group in each row of the original matrix, write each group including intermediate groups with all-zero entries into a separate row until the last non-zero group is reached.

- STEP 3: Link the symbol groups in the different rows originating from one row in the original matrix by inserting a new dedicated binary state variable which is placed between the neighboring symbol groups $i'$ and $i' + 1$. This state variable is then labelled with the state group index $l = i'$.

- STEP 4: Repeat STEPS 2 and 3 for all rows in the original matrix.

STEP 2 expands the original matrix in vertical direction and STEP 3 expands it in horizontal direction. The use of dedicated state variables in STEP 3 implies a column weight of two for each binary state variable. The total number of state variables in group $l$ (between neighboring symbol variable groups $i'$ and $i'+1$) depends on the number of rows in the original matrix with overlapping group span. This determines the alphabet of state variable $\boldsymbol{S}_l$ and thus the state complexity in the trellis representation of the code. Note that STEP 1 implies $\mathcal{X}_{i'} = (\mathbb{F}_2)^n$. As we will see later, this determines the branch complexity in the trellis representation.

When we apply the above algorithm to the example of the minimum span matrix for the (8,4,4) extended Hamming code in (3.7) and use duo-binary symbol variables with $n = 2$, we obtain the $10 \times 14$ matrix

| $\boldsymbol{x}_1 =$ | | $\boldsymbol{s}_1 =$ | | $\boldsymbol{x}_2 =$ | | $\boldsymbol{s}_2 =$ | | $\boldsymbol{x}_3 =$ | | $\boldsymbol{s}_3 =$ | | $\boldsymbol{x}_4 =$ | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $(x_1$ | $x_2)$ | $(s_1^{(1)}$ | $s_1^{(2)})$ | $(x_3$ | $x_4)$ | $(s_2^{(1)}$ | $s_2^{(2)})$ | $(x_5$ | $x_6)$ | $(s_3^{(1)}$ | $s_3^{(2)})$ | $(x_7$ | $x_8)$ |
| 1 | 1 | 1 | | | | | | | | | | | |
| | | 1 | | 1 | 1 | | | | | | | | |
| | | | | 1 | 1 | 1 | | | | | | | |
| | | | | | | 1 | | 1 | 1 | | | | |
| | | | | | | | | 1 | 1 | 1 | | | |
| | | | | | | | | | | 1 | | 1 | 1 |
| | 1 | | 1 | | | | | | | | | | |
| | | | 1 | | 1 | | 1 | | | | | | |
| | | | | | | | 1 | 1 | | | 1 | | |
| | | | | | | | | | | | 1 | 1 | |

(3.20)

with $\boldsymbol{x}_{i'} \in \mathcal{X}_{i'}$, $\mathcal{X}_{i'} = (\mathbb{F}_2)^2$ and $\boldsymbol{s}_l \in \mathcal{S}_l$, $\mathcal{S}_l = (\mathbb{F}_2)^2$. Each symbol variable $\boldsymbol{X}_{i'}$ now participates in exactly one local code $\mathcal{C}_j$ sharing the same index, i.e., $j = i'$. The local code $\mathcal{C}_j$ involves besides the symbol variable $\boldsymbol{X}_j$ also the preceding and succeeding state variables so that the local code words are defined according to (3.12) as $(\boldsymbol{x}_1, \boldsymbol{s}_1) \in \mathcal{C}_1$, $(\boldsymbol{s}_1, \boldsymbol{x}_2, \boldsymbol{s}_2) \in \mathcal{C}_2$, $(\boldsymbol{s}_2, \boldsymbol{x}_3, \boldsymbol{s}_3) \in \mathcal{C}_3$ and $(\boldsymbol{s}_3, \boldsymbol{x}_4) \in \mathcal{C}_4$. The local codes are fully described by the local parity-checks in (3.20) involving only the corresponding subset of symbol and state variables. The trellis representation of the code is then a graphical representation of all local codes and thus the set of all code words. Based on (3.20) we can easily derive the time-varying trellis representation of the (8,4,4) extended Hamming code with a state space dimension profile of (1,4,4,4,1) as shown in Fig. 3.5 a). The corresponding factor graph representation is shown in Fig. 3.5 b). Symbol and state variables are illustrated in the factor graph with circles and a double circles, respectively. Generalized constraint nodes are simply represented with a black box so that the local code and the complexity of the node is not revealed. Note that the two additional state variables $S_0$ and $S_4$ appear in Fig. 3.5. These state variables at the beginning and the end of the code graph are required because the conventional trellis representation assumes that all branches start in one state ($S_0$) and also end in one state ($S_4$). However, the inclusion of these state variables into (3.20) and the corresponding local codes $\mathcal{C}_1$ and $\mathcal{C}_4$ is dispensable since they only have a fixed value, i.e., $s_0 = 0$ and $s_4 = 0$. Due to the use of a matrix with minimum (group) span we obtain a trellis representation with a minimum number of trellis states, i.e., the minimal trellis of the code.

Note that the factor graph of a trellis representation naturally features constraint nodes with degree three while symbol and state variables typically have degree one and two, respectively.

### 3.3.3 Tailbiting Realizations

The state complexity of conventional trellis representations of block codes can in general be reduced by using tailbiting trellises. In a tailbiting representation of a code the beginning and the end of the code graph are connected together so that it forms a loop. In order to derive such

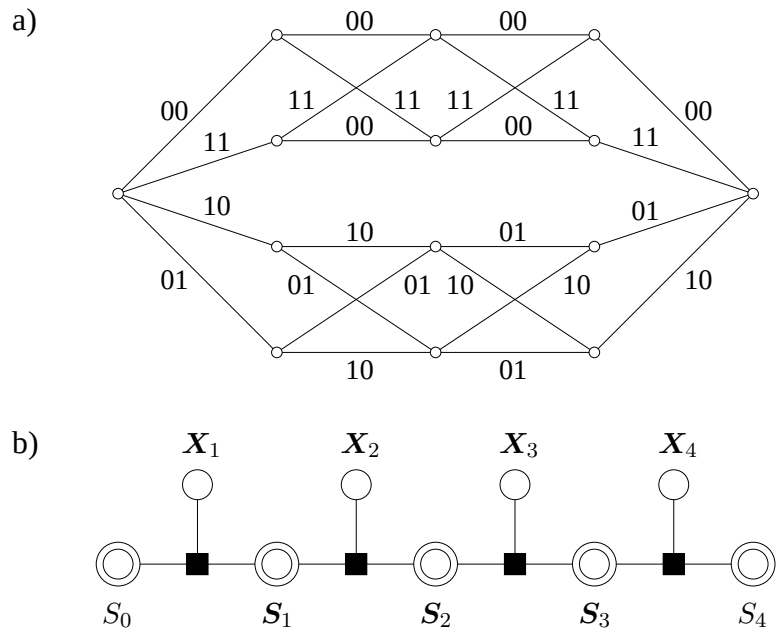



**Figure 3.5**: Minimal trellis representation of the (8,4,4) extended Hamming code with a state space dimension profile of (1,4,4,4,1) in a) and the associated factor graph in b).

a tailbiting trellis we modify the definition of a minimum span matrix as given in Section 3.2. Instead of defining the span as the index of the rightmost nonzero entry in the matrix minus the index of the leftmost nonzero entry we now work with a circular definition of the span. The overall cyclic span of a matrix may be smaller than the overall span thus leading to a trellis representation with a more favorable state space dimension profile. In case symbol variables are grouped together the span is replaced with the corresponding group span. The loop structure of the code graph allows the definition of local codes across the tailbiting joint. This typically reduces the state complexity near the middle of the conventional trellis.

Again, we use the example of the (8,4,4) extended Hamming code with the minimum span matrix in (3.7) in order to derive a tailbiting trellis representation of the code. When we group two symbol variables together into duobinary symbol variables, i.e., $n = 2$, we notice that both the group span and the cyclic group span in (3.7) is six. The matrix therefore also exhibits a minimum cyclic group span. When we apply the algorithm from Section 3.3.2 with the cyclic definition of the group span to (3.7) we obtain the $11 \times 15$ matrix

| | $\boldsymbol{x}_1 =$ | | $\boldsymbol{s}_1 =$ | | $\boldsymbol{x}_2 =$ | | | $\boldsymbol{x}_3 =$ | | $\boldsymbol{s}_3 =$ | | $\boldsymbol{x}_4 =$ | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $s_0$ | $(x_1$ | $x_2)$ | $(s_1^{(1)}$ | $s_1^{(2)})$ | $(x_3$ | $x_4)$ | $s_2$ | $(x_5$ | $x_6)$ | $(s_3^{(1)}$ | $s_3^{(2)})$ | $(x_7$ | $x_8)$ | $s_0$ |
| | 1 | 1 | 1 | | | | | | | | | | | |
| | | | 1 | | 1 | 1 | | | | | | | | |
| | | | | | 1 | 1 | 1 | | | | | | | |
| | | | | | | | 1 | 1 | 1 | | | | | |
| | | | | | | | | | | 1 | | | | |
| | | | | | | | | | | 1 | | 1 | 1 | |
| | | | | | | | | 1 | | | 1 | | | |
| | | | | | | | | | | | 1 | 1 | | 1 |
| 1 | | | | | | | | | | | | | | 1 |
| 1 | | 1 | | 1 | | | | | | | | | | |
| | | | | 1 | | 1 | | | | | | | | |

(3.21)

The first six rows in (3.21) originate from the first three rows in (3.7) and are essentially identical

to (3.20). The forth row in (3.7) now exploits the loop structure so that the first and second symbol group are connected to the third and fourth symbol group through the additional state variable $S_0$ at the tailbiting joint. Note that state variable $S_0$ appears twice in (3.21) and equality is achieved with a degree two parity-check. The local codes $\mathcal{C}_j$ involve, besides the symbol variable $X_j$, also the preceding and succeeding state variables so that the local code words are defined according to (3.12) as $(s_0, \boldsymbol{x}_1, \boldsymbol{s}_1) \in \mathcal{C}_1$, $(\boldsymbol{s}_1, \boldsymbol{x}_2, s_2) \in \mathcal{C}_2$, $(s_2, \boldsymbol{x}_3, \boldsymbol{s}_3) \in \mathcal{C}_3$ and $(\boldsymbol{s}_3, \boldsymbol{x}_4, s_0) \in \mathcal{C}_4$. The local codes are fully described by the local parity-checks in (3.21) involving only the corresponding subset of symbol and state variables. The cyclic structure allows the use of a binary state variable $S_2$ in (3.21) instead of the duo-binary state variable in (3.20) at the expense of an additional binary state variable $S_0$. This leads to a minimal tailbiting trellis representation with 4 trellis sections and a state space dimension profile of (2,4,2,4,2) as illustrated in Fig. 3.6 a). Such a minimal tailbiting trellis was already presented in [CFV99] for a different bit ordering. The corresponding factor graph representation is shown in Fig. 3.6 b).

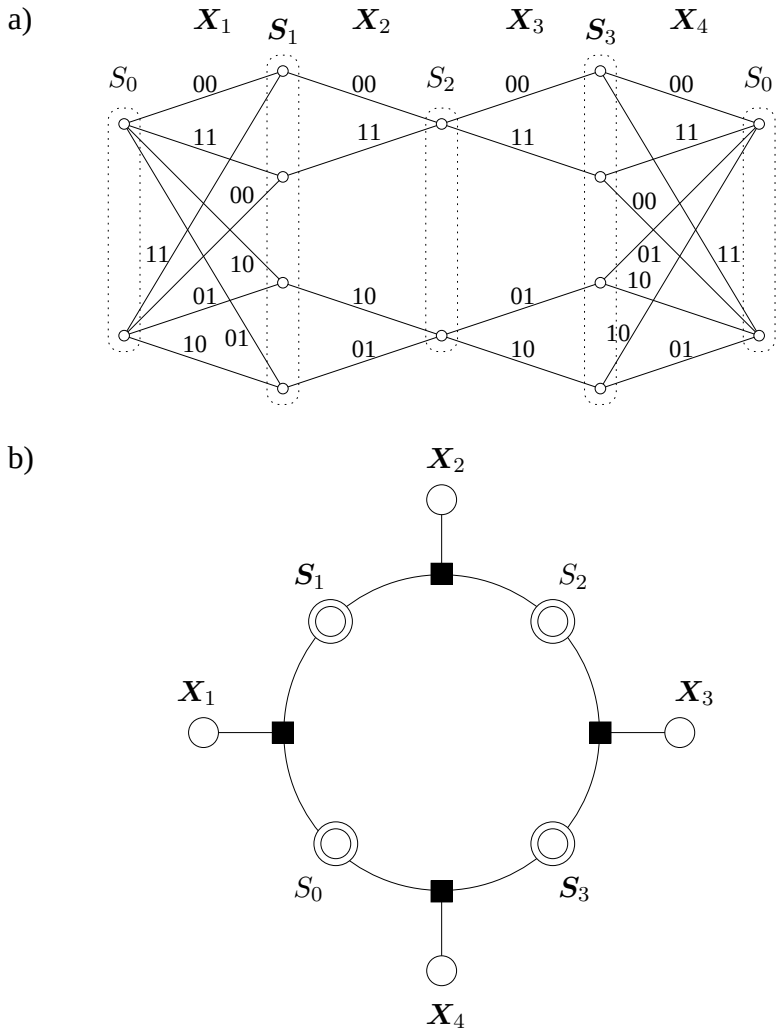



**Figure 3.6**: Minimal tailbiting trellis representation of the (8,4,4) extended Hamming code with a state space dimension profile of (2,4,2,4,2) in a) and the associated factor graph in b).

## 3.4 Normal Graphs

Normal graphs were derived from factor graphs by Forney [For01]. One of the main differences lies in the representation of variables in the code graph. Factor graphs represent variables as nodes while normal graphs use edges. The usage of edges implies that variables can only participate in up to two constraint nodes. We differentiate between (visible) symbol variables and (hidden or internal) state variables. A symbol variable is represented by a half edge or a cut edge similar to an input/output (I/O) port. Symbol variables thus only participate in a single constraint node. This is in contrast to state variables which are represented as ordinary edges between two local constraint nodes. Half edges can directly be connected to other half edges thus generating ordinary edges. Similarly, an ordinary edge can be cut into two half edges. The key advantage of normal graphs is that all computations associated with the local codes concentrate in the corresponding nodes while edges are solely used for the communication between the nodes. The nodes can then be interpreted as local node processors and the edges become the connecting wires between them. This special characteristic of normal graphs forms the basis for the representation of FEC decoders in the subsequent chapters.

In the following we describe how factor graphs are converted into normal graphs. This conversion introduces new equality constraint nodes which are then also transformed into nodes with a degree of no more than three as outlined in Section 3.3.

### 3.4.1 Conversion of Factor Graphs

A factor graph can easily be converted into the corresponding normal graph representation using the conversion rules in Fig. 3.7. Whenever a symbol variable $X_i$ is involved in more

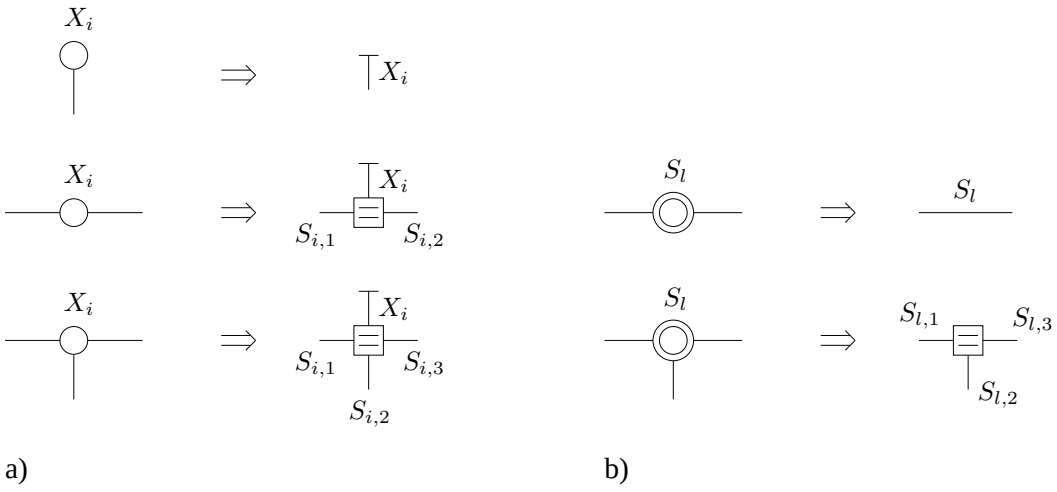



**Figure 3.7**: Conversion of symbol and state variable nodes in factor graphs into the corresponding normal graph representation.

than one constraint node or a state variable $S_l$ participates in more than two constraint nodes, the corresponding variables are duplicated and connected to the original variables through an equality constraint node. Such an equality constraint node forces all variables, i.e., edges, to be identical. It is thus illustrated in the normal graph as rectangle with an equality sign inside as shown in Fig. 3.7. Each symbol variable is then only connected to one constraint node. State variables are connected to exactly two constraint nodes, i.e., the newly introduced equality constraint node and the original constraint node. In general, each symbol variable $X_i$ with degree $d_{v,i} > 1$ in the factor graph is duplicated by $d_{v,i}$ state variables $S_{i,\nu} = X_i$ with $\nu \in \{1, \dots d_{v,i}\}$. Similarly, each state variable $S_l$ with degree $d_{v,l} > 2$ in the factor graph is replaced

by $d_{v,l}$ new state variables $S_{l,\xi} = S_l$ with $\xi \in \{1, \ldots, d_{v,l}\}$. No equality constraint nodes are necessary for degree one symbol variables and degree two state variables. In this case, the symbol variables simply become half edges and the state variables translate into ordinary edges. Generalized constraint nodes in factor graphs do not require any conversion. They are therefore also represented in the normal graph with a black box. For the special case of the exclusive OR (XOR) operation of binary variables we use $\boxplus$ instead of $\oplus$.

When we apply the conversion rules from Fig. 3.7 to the binary factor graph for the (8,4,4) extended Hamming code in Fig. 3.4 we obtain the associated normal graph in Fig. 3.8. A simi-

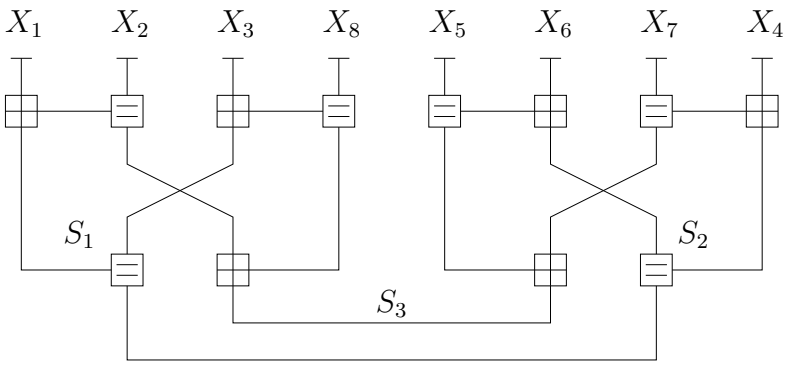



**Figure 3.8**: Normal graph representation of the (8,4,4) extended Hamming code with minimum girth six.

lar code graph can be found in [For01]. The two non-binary factor graphs for (8,4,4) extended Hamming code presented in Fig. 3.5 and Fig. 3.6 can also directly be converted into normal graphs. The state variables $S_0$ and $S_4$ in Fig. 3.5 are special cases since they are connected to only one constraint node. These state variables can either be represented in the normal graph with a half edge or, alternatively, annexed to the neighboring constraint nodes since they have only one fixed state.

### 3.4.2 Degree Restriction on Nodes

In this section we transform normal graphs in a way that the degree of all nodes in the code graph is three. In the following we focus on binary code graphs based on the parity-check matrix of the code. It is assumed that the rows and columns of the parity-check matrix $\boldsymbol{H}$ are indexed with $j$ and $i$, respectively, and that row $j$ of the parity-check matrix has a row weight of $d_{c,j}$ and that column $i$ has a column weight of $d_{v,i}$. This notation is consistent with Section 2.3.2. The $j$-th row of the parity-check matrix translates into a parity-check node in the normal graph with degree $d_{c,j}$. Here, the degree of a node is defined as the number of edges connected to it. The columns of the parity-check matrix represent equality nodes of degree $d_{v,i}+1$ in the normal graph. The difference between the column weight and the degree of the corresponding equality node is due to the additional half edge used for symbol variables in the normal graph. Note that this differs from the degree of symbol variable nodes in Tanner graphs or factor graphs.

Prominent examples of codes based on a binary graphs are LDPC codes. The normal graph of a generic LDPC code based on the $(N-K) \times N$ parity-check matrix of the code is shown in Fig. 3.9. On the left hand side of the code graph we observe $K$ binary symbol variables $X_i$, $i \in \{1, \ldots, N\}$ which are duplicated $d_{v,i}$ times in the corresponding equality constraint nodes. On the right hand side of the code graph we have a total number of $N-K$ parity-check constraint nodes where constraint node $j$, $j \in \{1, \ldots, N-K\}$ has degree $d_{c,j}$. The total number

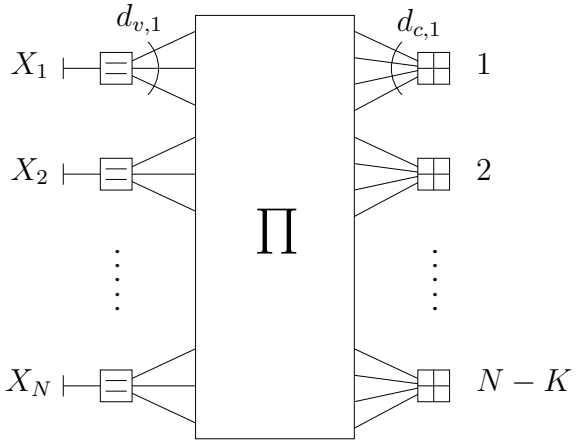



**Figure 3.9**: Normal graph representation of a generic LDPC code.

of branches (i.e., state variables) in the interleaver network $\Pi$ is then determined by

$$\sum_{i=1}^{N} d_{v,i} = \sum_{j=1}^{N-K} d_{c,j}. \tag{3.22}$$

The transformation of the normal graph in Fig. 3.9 into a normal graph with degree three nodes proceeds as follows. Every check node with degree $d_{c,j} > 3$ is split up into $d_{c,j} - 2$ check nodes of degree three. This can be achieved by applying the C3 Algorithm from Section 3.3.1 to the parity-check matrix of the code. Here, we assume that the C3 Algorithm does not generate state variables with a column weight of more than three. Similarly, every equality constraint node for symbol variables with more than three edges connected to it, i.e., $d_{v,i} > 2$, is split up into $d_{v,i} - 1$ equality constraint nodes with degree three. This degree limitation of equality constraint nodes can be accomplished with the V3 Algorithm which is similar to the C3 Algorithm in Section 3.3.1.

The V3 Algorithm is divided into three steps:

- STEP 1: Identify all columns in the matrix corresponding to symbol variables with $d_{v,i} > 2$, $i \in I_{\mathcal{X}}$.
- STEP 2: Add $d_{v,i} - 2$ new columns for each column identified in STEP 1. Distribute the $d_{v,i}$ ones in the original column over these $d_{v,i} - 1$ columns in a way that the first (i.e., the original column) and the following $d_{v,i} - 3$ new columns have weight one and only the last column has weight two. Maintain the row positions of the ones.
- STEP 3: Add $d_{v,i} - 2$ new rows for each column identified in STEP 1. Add two ones in each of these rows so that all the $d_{v,i} - 1$ columns obtained from one column in the original matrix are linked pairwise in sequential order, i.e., the 1st and the 2nd, the 2nd and the 3rd, and so forth. The original column then has weight two and all new columns generated in STEP 2 have weight three.

Note that STEP 2 and 3 expand the matrix in horizontal and vertical direction, respectively. The column weight for symbol variables is then two and the column weight for state variables is limited to three. Hence, the degree of all equality constraint nodes in the normal graph is limited to three. Step 2 and 3 can be reversed by adding up all columns which originated from one column in the original matrix. This yields the original columns and all rows added in STEP

3 have only zero entries, i.e., the corresponding degree two parity-checks become redundant.

When we apply both the C3 Algorithm from Section 3.3.1 and the above V3 Algorithm to the parity-check matrix of a LDPC code we obtain a matrix representation of the transformed normal graph. An example of such a transformed normal graph of a generic LDPC code is depicted in Fig. 3.10. Note that all equality constraint nodes and parity-check constraint nodes feature degree three.

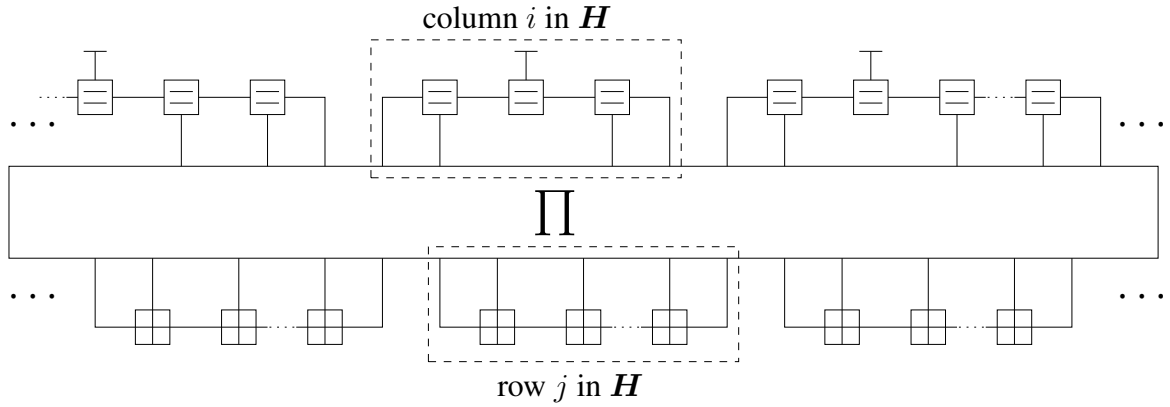



**Figure 3.10**: Transformed normal graph of a generic LDPC code with degree three nodes.

# 4

# *Decoding Based on Graphs*

Many popular decoding algorithms can be interpreted as message passing decoding based on code graphs. Several different types of code graphs were covered in Chapter 3. In the context of this thesis we prefer normal graphs because of their similarity to block diagrams of FEC decoders. Normal graphs will be heavily used in the next chapters in order to represent our analog decoding networks. All computations of the decoder are concentrated in the nodes of the normal graph while the edges are solely used for the communication between adjacent node processors. Half edges for symbol variables are used for the input of channel values and the output of the decoding result. Depending on the structure of the normal graph, the used message format, the local computations in the node processors and the scheduling of the messages it is possible to view various known decoding algorithms as message passing on normal graphs. These include the sum-product and min-sum (or max-sum) algorithms, the Viterbi algorithm [Vit67], the BCJR algorithm [BCJR74] and iterative decoding of LDPC codes [Gal62] or turbo codes [BGT93]. The message passing algorithm is exact, i.e., optimal, only on cycle-free code graphs, but there are many cases in which the message passing algorithm gives remarkably good results even in cases where the code graph has loops. In general, decoding based on code graphs with loops allows the approximation of optimal decoding with a reduced decoder complexity. A code graph with cycles can be split up into two or more possibly cycle-free graph fragments which are decoded separately. The information exchange between these graph fragments then occurs in several iterations. Typical examples are iterative decoders for LDPC codes or turbo codes. The code graph may also be split up into several segments which are processed in the decoder in a sequential fashion in order to reduce decoding delay and storage requirements. An example of this is sliding window decoding of convolutional codes.

We start with the formulation of the general decoding problem in Section 4.1 and then introduce the general message passing decoding algorithm in Section 4.2. We differentiate between message passing decoding based on binary and non-binary code graphs which are covered in Section 4.3 and 4.4, respectively. Different optimal and suboptimal decoding algorithms are considered. Section 4.5 summarizes a powerful technique for the analysis of iterative decoding based on mutual information. In Section 4.6 we then pay attention to the quantization of soft information.

## 4.1 General Decoding Problem

For the formulation of the general decoding problem we assume that a vector of information bits $\boldsymbol{u}$ is encoded into a vector of code bits $\boldsymbol{c} \in \mathcal{C}$ and transmitted over a noisy communication channel. At the receiver we then acquire the corresponding vector $\boldsymbol{y}$ of sampled matched filter outputs, which we simply refer to as channel values. The task of the channel decoder can be defined as finding the most likely code word $\hat{\boldsymbol{c}}$ which minimizes the word or block error probability

$$P_w = \sum_{\boldsymbol{y}} P(\hat{\boldsymbol{c}} \neq \boldsymbol{c}|\boldsymbol{y})P(\boldsymbol{y}) = \sum_{\boldsymbol{y}} P(\hat{\boldsymbol{u}} \neq \boldsymbol{u}|\boldsymbol{y})P(\boldsymbol{y}), \tag{4.1}$$

with $P(\boldsymbol{y})$ as the probability of the received vector of channel values. Note that the estimated information bits $\hat{\boldsymbol{u}}$ can directly be obtained from the estimated code bits $\hat{\boldsymbol{c}}$ using the inverse encoder mapping. For a given vector $\boldsymbol{y}$ it is sufficient to minimize $P(\hat{\boldsymbol{c}} \neq \boldsymbol{c}|\boldsymbol{y})$ or equivalently maximize $P(\hat{\boldsymbol{c}}|\boldsymbol{y})$ by finding the most likely code word $\hat{\boldsymbol{c}}$ given that $\boldsymbol{y}$ is received. A decoder decision based on this criterion represents the optimal solution to the general decoding problem and the corresponding decoder is then called a maximum a posteriori (MAP) sequence decoder. With Bayes' rule we obtain

$$P(\hat{\boldsymbol{c}}|\boldsymbol{y}) = \frac{P(\boldsymbol{y}|\hat{\boldsymbol{c}})P(\hat{\boldsymbol{c}})}{P(\boldsymbol{y})}. \tag{4.2}$$

When we assume equally likely code words we find that maximizing $P(\hat{\boldsymbol{c}}|\boldsymbol{y})$ is equivalent to maximizing $P(\boldsymbol{y}|\hat{\boldsymbol{c}})$. A decoder which uses this criteria for making a decoder decision is called maximum likelihood (ML) sequence decoder. In this case the ML decoder leads to the same word or block error probability as the MAP decoder. Note that ML decoding is equivalent to minimum distance (MD) decoding which chooses the code word $\hat{\boldsymbol{c}}$ which is closest to the received vector $\boldsymbol{y}$ in terms of the Hamming distance. A decoder based on the above rules is also referred to as a sequence estimator since it finds the sequence of code bits, i.e., a valid code word, which was most likely transmitted over the communication channel. In many cases it is desirable to minimize the (information) symbol error probability rather than the word or block error probability. Similar to the above, the symbol-by-symbol a posteriori probability (APP)

$$P(u_k|\boldsymbol{y}) \tag{4.3}$$

represents the probability of information bit $u_k \in \mathbb{F}_2$, $k \in I_{\mathcal{U}}$, given the received vector of channel values $\boldsymbol{y}$. The optimum symbol estimator then finds the most likely symbol $\hat{u}_k$ according to

$$\hat{u}_k = \arg\max_{u_k \in \mathbb{F}_2} P(u_k|\boldsymbol{y}). \tag{4.4}$$

This decision criterion is known as the symbol-by-symbol MAP decoding rule and the decoder which outputs $\hat{u}_k$ is referred to as the symbol-by-symbol MAP decoder. The difference between the MAP decoder and the APP decoder is that the APP decoder not only provides the hard decision $\hat{u}_k$ but also some reliability information about this bit decision, e.g., $P(\hat{u}_k|\boldsymbol{y})$. Let us assume that every information bit $u_k$ corresponds to bit $x_i$ in the vector of possible symbol configurations $\boldsymbol{x} \in \mathcal{X}$ with $\mathcal{C} \subseteq \mathcal{X}$ and $i \in I_{\mathcal{X}}$. This means that we have a systematic code where the information bits appear among the code bits. The APP decoder output in (4.3) can then be expressed as

$$P(u_k|\boldsymbol{y}) = P(x_i|\boldsymbol{y}) = \frac{P(x_i, \boldsymbol{y})}{P(\boldsymbol{y})} = \sum_{\boldsymbol{x} \in \mathcal{C}, x_i} \frac{P(\boldsymbol{y}|\boldsymbol{x})P(\boldsymbol{x})}{P(\boldsymbol{y})}. \tag{4.5}$$

It is convenient to express the APP decoder output in terms of the log-likelihood ratio [HOP96]

$$L(\hat{X}_i) = \ln \frac{P(x_i=0|\boldsymbol{y})}{P(x_i=1|\boldsymbol{y})} = \ln \frac{\sum\limits_{\boldsymbol{x}\in\mathcal{C},x_i=0} P(\boldsymbol{y}|\boldsymbol{x})P(\boldsymbol{x})}{\sum\limits_{\boldsymbol{x}\in\mathcal{C},x_i=1} P(\boldsymbol{y}|\boldsymbol{x})P(\boldsymbol{x})} \tag{4.6}$$

$$= \ln \frac{\sum\limits_{\boldsymbol{x}\in\mathcal{C},x_i=0} \prod\limits_{\eta=1}^{|I_{\mathcal{X}}|} p(y_\eta|x_\eta)P(x_\eta)}{\sum\limits_{\boldsymbol{x}\in\mathcal{C},x_i=1} \prod\limits_{\eta=1}^{|I_{\mathcal{X}}|} p(y_\eta|x_\eta)P(x_\eta)} \tag{4.7}$$

$$= \underbrace{\ln \frac{p(y_i|x_i=0)}{p(y_i|x_i=1)}}_{L_c y_i} + \underbrace{\ln \frac{P(x_i=0)}{P(x_i=1)}}_{L_a(X_i)} + \underbrace{\ln \frac{\sum\limits_{\boldsymbol{x}\in\mathcal{C},x_i=0} \prod\limits_{\eta=1,\eta\neq i}^{|I_{\mathcal{X}}|} p(y_\eta|x_\eta)P(x_\eta)}{\sum\limits_{\boldsymbol{x}\in\mathcal{C},x_i=1} \prod\limits_{\eta=1,\eta\neq i}^{|I_{\mathcal{X}}|} p(y_\eta|x_\eta)P(x_\eta)}}_{L_e(X_i)}. \tag{4.8}$$

Note that the common factor $1/P(\boldsymbol{y})$ in (4.5) cancels out in when we use log-likelihood ratios. The APP decoder output in (4.8) can be simply expressed as the sum of the weighted channel value $L_c y_i$, the a priori information $L_a(X_i)$ and the extrinsic information $L_e(X_i)$ according to

$$L(\hat{X}_i) = L_c y_i + L_a(X_i) + L_e(X_i). \tag{4.9}$$

The sign of $L(\hat{X}_i)$ is equivalent to the hard decision of the symbol-by-symbol MAP decoder and the magnitude $|L(\hat{X}_i)|$ provides a reliability measure for the bit decision. In case there is no a priori information available we gain $P(x_i=0) = P(x_i=1) = 0.5$ and thus $L_a(X_i) = 0$.

For many interesting coding schemes including LDPC codes and turbo codes the exact calculation of the APP decoder output is computational intractable. In this case we need to rely on suboptimal solutions to the decoding problem as discussed later in this chapter.

## 4.2 General Message Passing Decoding Algorithm

Many popular decoding algorithms can be seen as instances of a general message passing decoding algorithm which operates on normal graphs. We differentiate between different realizations of the message passing decoding algorithm based on the following properties:

- **Structure of the code graph**
  The structure of the normal graph determines the paths for the bidirectional message exchange in the decoder without specifying the message format, the local computations in the node processors and the scheduling of the messages. There are various different normal graphs for a given code which form the basis for different decoding algorithms. In general, a different structure of the code graph yields a different decoder performance. A cycle-free normal graph facilitates optimal decoder performance while a normal graph with cycles implies a suboptimal decoder performance. However, code graphs with loops may lower the computational complexity in the associated decoder.

- **Message representation**
  The messages which are passed along the edges of the normal graph may have various different message formats. This includes likelihoods, log-likelihoods, log-likelihood ratios and soft bits. A message may consist of a single real number or a vector of real numbers

depending on the message format and the alphabet of the associated variables in the normal graph. In a digital decoder implementation the messages need to be represented with a finite number of quantization levels.

- **Computations in the node processors**
  The computations in the node processors are determined by the associated local codes and the selected message format. When the messages are represented by likelihood vectors the node processors perform multiplications and summations and we obtain the well known sum-product algorithm. Digital decoder implementations typically use vectors of log-likelihoods (or negative log-likelihoods) in order to replace the multiplication of messages with summations, which are easier to implement. It is also common practice in digital designs to replace the exact decoding operations in the node processors with approximations in order to reduce decoder complexity [RVH95]. A typical example is the suboptimal max-sum (or min-sum) algorithm. A special case of this algorithm is the well known Viterbi algorithm [Vit67] for ML decoding which performs these computations only in one direction[1] along the normal graph representing the trellis.

- **Scheduling of the computations and the message exchange**
  There are various different schedules for the computation of the messages and the message exchange on the normal graph. At the beginning of the decoding process all half edges belonging to symbol variables are initialized according to the received channel values and all state variables are initialized so that all possible states are equiprobable. A simple schedule may for example require that a node processor calculates or updates its output whenever there is a new message on at least one of the corresponding inputs. The output is then propagated along an edge in the normal graph to a connected node processor where it then triggers another local computation. Such a schedule typically leads to more computations than necessary so that it is in general not very efficient. This is because a new message on one of the inputs may not alter the output of a generalized constraint node processor, e.g., when another input belonging to a state variable is still initialized as described in the above. This schedule can be modified in a way that the output of a node processor is calculated or updated only when all inputs associated with a state variable provide new messages. Based on a cycle-free normal graph this leads to a very efficient decoding algorithm since the messages only need be computed once for each direction of the edges. When the normal graph represents the conventional code trellis this scheduling leads to a message exchange in both forward and backward direction as dictated by the BCJR algorithm [BCJR74]. Decoding then naturally terminates as soon as all node processors have calculated their outputs. Decoding based on normal graphs with loops is only suboptimal and lacks such a clear termination. An example of this is a decoder based on the normal graph of a tailbiting trellis representation of the code. Here, the messages propagate in forward and backward direction around the tailbiting ring as described by the tailbiting APP algorithm [AH98]. Decoding then stops after a certain number of wraps around the tailbiting ring. Another example is iterative decoding of LDPC codes [Gal62]. In this case, the decoder works with a very similar schedule which distinguishes between equality node processors and check node processors. In the first half of an iteration all equality node processors compute their outputs which are then passed on to the check node processors. The check node processors then calculate the new messages for the equality node processors in the second half of the iteration. This iterative message exchange continues for a certain number of iterations, or, until another stopping criteria is fulfilled, e.g., a valid code word has been found. Normal graphs of turbo codes also exhibit loops. Decoding of turbo codes [BGT93] relies on two (or more) component decoders which exchange their messages in several iterations similar to a LDPC decoder.

[1] An additional trace back for the surviving path is required.

It is somehow surprising that message passing decoding of LDPC codes and turbo codes allows a remarkable good decoder performance close to the Shannon limit despite the presence of cycles in the code graph.

In the following sections we cover different decoding algorithms based on binary and non-binary normal graphs in more detail.

## 4.3 Decoding based on Binary Graphs

We start with message passing decoding based on binary code graphs. Binary code graphs are typically obtained from a parity-check matrix (or an extended parity-check matrix) of the code. The most prominent example of codes based on binary code graphs are LDPC codes. Here, the code graph has loops of different size so that message passing decoding becomes suboptimal and iterative. However, iterative decoding of these class of codes exhibits excellent error correcting performance and channel capacity may be closely approached. The associated decoder consists of equality node processors and check node processors. In the following we cover different message representations for binary random variables and the associated computations in the node processors of the decoder. We then summarize iterative decoding of LDPC codes based on normal graphs.

### 4.3.1 Message Representations for Binary Random Variables

A binary random variable $X$ with the two possible outcomes $x \in \mathbb{F}_2$ can be described by the two probabilities $P_X(x = 0)$ and $P_X(x = 1)$ with $P_X(x = 0) + P_X(x = 1) = 1$. The log-likelihood of $P_X(x)$ is defined as $\ln(P_X(x)) = l_x(X)$, where $\ln$ denotes again the natural logarithm. Binary random variables can then be described with the probability vector

$$( P_X(0), P_X(1) ), \tag{4.10}$$

or the vector of log-likelihoods

$$( \ln(P_X(0)), \ln(P_X(1)) ) = ( l_0(X), l_1(X) ). \tag{4.11}$$

For some message representations it is convenient to work with antipodal realizations of binary random variables, i.e., $x \in \{+1, -1\}$. Here, we always assume the mapping $0 \leftrightarrow +1$ and $1 \leftrightarrow -1$ which is consistent with the mapping used for the BPSK modulation of the transmitted signals in Section 2.2. In the following we will no longer distinguish between $x \in \{0, 1\}$ and $x \in \{+1, -1\}$. We can then use a single real number, e.g., the likelihood ratio

$$LR(X) = \frac{P_X(+1)}{P_X(-1)}, \tag{4.12}$$

or, the log-likelihood ratio

$$\begin{aligned} L(X) &= \ln \frac{P_X(+1)}{P_X(-1)}, & (4.13)\\ &= l_{+1}(X) - l_{-1}(X), & (4.14) \end{aligned}$$

in order to fully characterize the binary random variable. The log-likelihood ratio $L(X)$ is also referred to as the L-value of variable $X$. The sign of $L(X)$ represents the hard decision for the bit, i.e., $\hat{x} = \mathrm{sign}(L(X))$, while the magnitude $|L(X)|$ represents a measure for the reliability of the bit decision. The larger the magnitude the more reliable is the bit decision. Note that the log-likelihood ratio at the output of the APP decoder in (4.9) is composed of three independent

L-values. Alternatively, random variable $X$ is also fully described by the soft bit $\lambda(X)$, which is defined as the expectation of the random variable according to

$$\begin{aligned}\lambda(X) &= \mathrm{E}\{X\} = (+1)P_X(+1) + (-1)P_X(-1), \\ &= \frac{e^{L(X)}-1}{e^{L(X)}+1}, \\ &= \tanh\left(\frac{L(X)}{2}\right). \end{aligned} \tag{4.15}$$

Again, we have $\hat{x} = \mathrm{sign}(\lambda(X))$ with $|\lambda(X)|$ as reliability measure for the bit decision. This non-linear transformation from the L-value $L(X)$ to the soft bit $\lambda(X)$ limits the range of the messages to values between $-1$ and $+1$. In the following we use a simplified notation for the probabilities where the index of the random variable $X$ is skipped whenever there is no danger of confusion.

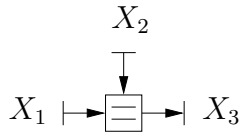
The conversion from one message representation to another message representation is summarized in Fig. 4.1.

| from \ to | $P(x)$ | $\lambda(X)$ | $LR(X)$ | $L(X)$ |
|---|---|---|---|---|
| $P(x)$ | —— | $\lambda(X) = P(x=+1) - P(x=-1)$ | $LR(X) = \frac{P(x=+1)}{P(x=-1)}$ | $L(X) = \ln \frac{P(x=+1)}{P(x=-1)}$ |
| $\lambda(X)$ | $P(x=+1) = \frac{1+\lambda(X)}{2}$<br>$P(x=-1) = \frac{1-\lambda(X)}{2}$ | —— | $LR(X) = \frac{1+\lambda(X)}{1-\lambda(X)}$ | $L(X) = \ln \frac{1+\lambda(X)}{1-\lambda(X)}$<br>$= 2\tanh^{-1}(\lambda(X))$ |
| $LR(X)$ | $P(x=+1) = \frac{LR(X)}{1+LR(X)}$<br>$P(x=-1) = \frac{1}{1+LR(X)}$ | $\lambda(X) = \frac{LR(X)-1}{1+LR(X)}$ | —— | $L(X) = \ln(LR(X))$ |
| $L(X)$ | $P(x=+1) = \frac{e^{L(X)}}{1+e^{L(X)}}$<br>$P(x=-1) = \frac{1}{1+e^{L(X)}}$ | $\lambda(X) = \frac{e^{L(X)}-1}{e^{L(X)}+1}$<br>$= \tanh\left(\frac{L(X)}{2}\right)$ | $LR(X) = e^{L(X)}$ | —— |

**Figure 4.1**: Conversion table for different message representations.

### 4.3.2 Binary Constraint Node Processors

Binary normal graphs consist of edges representing binary random variables and two types of constraint nodes, i.e., equality constraint nodes and parity-check constraint nodes. Fig. 4.2 depicts these two types of node processors with degree three and a message flow exposed towards random variable $X_3$. Depending on the used message format we obtain different computations inside the node processors. The corresponding computations are listed in Fig. 4.2 in terms of likelihood vectors, soft bits $\lambda(X_i)$, likelihood ratios $LR(X_i)$ and log-likelihood ratios $L(X_i)$ with $i \in \{1,2,3\}$ and $x_i \in \{0,1\}$. On binary code graphs we pay particular attention to message representations in terms of log-likelihood ratios. Here, a processor for an equality constraint node performs the summation of L-values while the processor for a parity-check constraint node involves the tangent hyperbolic and the inverse tangent hyperbolic functions as listed in Fig. 4.2. This nonlinear operation is commonly referred to as the Boxplus operation of L-values which is abbreviated in the following with the $\boxplus$ symbol [HOP96].

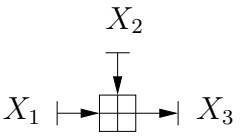



$$\begin{pmatrix} P(x_3=0) \\ P(x_3=1) \end{pmatrix} = \begin{pmatrix} \frac{P(x_1=0)P(x_2=0)}{P(x_1=0)P(x_2=0)+P(x_1=1)P(x_2=1)} \\ \frac{P(x_1=1)P(x_2=1)}{P(x_1=0)P(x_2=0)+P(x_1=1)P(x_2=1)} \end{pmatrix}$$

$$\lambda(X_3) = \frac{\lambda(X_1)+\lambda(X_2)}{1+\lambda(X_1)\lambda(X_2)}$$

$$LR(X_3) = LR(X_1)LR(X_2)$$

$$L(X_3) = L(X_1) + L(X_2)$$




$$\begin{pmatrix} P(x_3=0) \\ P(x_3=1) \end{pmatrix} = \begin{pmatrix} P(x_1=0)P(x_2=0) + P(x_1=1)P(x_2=1) \\ P(x_1=0)P(x_2=1) + P(x_1=1)P(x_2=0) \end{pmatrix}$$

$$\lambda(X_3) = \lambda(X_1)\lambda(X_2)$$

$$LR(X_3) = \frac{1+LR(X_1)LR(X_2)}{LR(X_1)+LR(X_2)}$$

$$L(X_3) = 2\tanh^{-1}\left[\tanh\left(\tfrac{L(X_1)}{2}\right)\tanh\left(\tfrac{L(X_2)}{2}\right)\right] = L(X_1) \boxplus L(X_2)$$

**Figure 4.2**: Computations in binary constraint node processors for different message representations.

### 4.3.3 Iterative Decoding of LDPC Codes

Iterative decoding of LDPC codes relies on binary constraint node processors. In the following we assume that the decoder is based on the normal graph of the $(N-K) \times N$ parity-check matrix $\boldsymbol{H}$ of code $\mathcal{C}$ with $j \in I_{\mathcal{C}}$ and $i \in I_{\mathcal{X}}$ as the row and column index of $\boldsymbol{H}$, respectively. The generic normal graph of a LDPC code in shown in Fig. 3.9. The associated decoding network consists of equality constraint nodes and parity-check constraint nodes which are connected to each other through an interleaver network. The corresponding node processors were already introduced in Section 4.3.2 for the case of nodes with degree three. These node processors will later be utilized in our analog decoders. In this section we do not impose any degree restrictions on the nodes in order to describe the iterative decoding process independent of a particular implementation. Decoding based on binary code graphs is conveniently described in terms of log-likelihood ratios. The equality node processor then simply performs the summation of incoming L-values while the check node processor performs the Boxplus operation of L-values, compare Fig. 4.2. The typical scheduling of the messages in the decoder dates back to Gallager [Gal62], [Gal63] and a brief summary is given below.

The LDPC decoder is divided into the set of equality node processors and the set of check node processors which form the two component decoders. The received vector of L-values $L_c\boldsymbol{y}$ with $\boldsymbol{y} = (y_1, \ldots, y_N)$ corresponding to the vector of code bits $\boldsymbol{x} = (x_1, \ldots, x_N)$ is stored

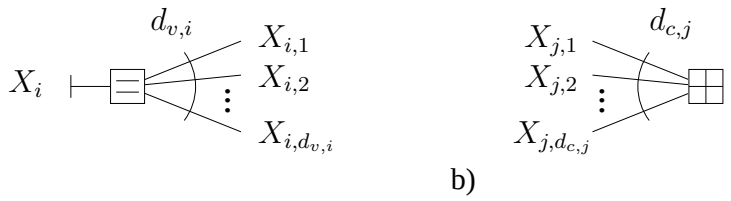



**Figure 4.3**: Normal graph of an equality node processor for symbol variables in a) and a check node processor in b).

at the input of the decoder. Furthermore, there is additional internal memory in order to store the extrinsic information generated and exchanged by the two component decoders. Due to the bidirectional message exchange on the normal graph there are two storage elements for each edge in the code graph. Note that the number of edges in the normal graph (with respect to the interleaver) is determined by the number of ones in the corresponding $\boldsymbol{H}$ matrix. At the beginning of the decoding process new channel values are loaded and all the extrinsic memory is reset to zero. The messages in the decoder are scheduled in a way that in the first half of an iteration every equality node processor $i$ with degree $d_{v,i}+1$ as shown in Fig. 4.3 a) calculates the outgoing extrinsic information on edge $\eta$ according to

$$L_e(X_{i,\eta}) = L_c y_i + \sum_{\xi=1,\xi\neq\eta}^{d_{v,i}} L_a(X_{i,\xi}), \tag{4.16}$$

with $\eta \in \{1,\ldots,d_{v,i}\}$. This calculation is based on the channel value $L_c y_i$ associated with symbol variable $X_i$ and the incoming a priori information $L_a(X_{i,\xi})$ obtained from the connected check node processors. Note that the incoming message from the edge on which the message is send out, i.e., $\xi=\eta$, is omitted in the calculation of (4.16). Further note that in the first iteration all the a priori information is zero so that only the channel values $L_c y_i$ are send to the check node processors. In the second half of the iteration every check node processor $j$ with degree $d_{c,j}$ as shown in Fig. 4.3 b) calculates the outgoing extrinsic information on edge $\nu$ according to

$$L_e(X_{j,\nu}) = \sum_{\xi=1,\xi\neq\nu}^{d_{c,j}} \boxplus\, L_a(X_{j,\xi}), \tag{4.17}$$

with $\nu \in \{1,\ldots,d_{c,j}\}$. Here, the incoming a priori information $L_a(X_{j,\xi})$ corresponds to the interleaved version of the extrinsic information as calculated by the equality node processors in the first half of the iteration. The extrinsic output of the check node processors is then de-interleaved and passed back to the equality node processors where it is used as a priori information in the next iteration. This iterative decoding process typically terminates after a predefined number of iterations, or, as soon as a valid code word has been found. The decoder output for symbol variable $X_i$ is then determined by

$$L(\hat{X}_i) = L_c y_i + \sum_{\xi=1}^{d_{v,i}} L_a(X_{i,\xi}). \tag{4.18}$$

The only difference compared to (4.16) is that now all the incoming a priori information contributes to the calculation of the decoder output.

There is an alternative method for the calculation of the extrinsic output $L_e(X_{i,\eta})$ of equality node processor $i$ on edge $\eta$. Instead of (4.16) we can first calculate the overall decoder output

according to (4.18) and then subtract the a priori information $L_a(X_{i,\eta})$ obtained from edge $\eta$, i.e.,

$$L_e(X_{i,\eta}) = L(\hat{X}_i) - L_a(X_{i,\eta}). \tag{4.19}$$

Depending on the degree of the equality constraint nodes this alternative computation may reduce decoder complexity. Furthermore, the calculation in (4.19) can be exploited by a modified message passing where the equality node processors only calculate the decoder output according to (4.18) which is then also passed on to all connected check node processors. The subtraction in (4.19) is then performed locally in the different check node processors. This technique was, e.g., adopted in [DCK05] in order to reduce the complexity of the interconnects between equality and check node processors.

There are further message passing schedules for LDPC codes which exhibit a special code structure. This includes layered message passing decoding of specially constructed LDPC codes [MS03], [Hoc04] and decoding based on the repeat accumulate structure of certain LDPC codes [DJM98], [JKM00], [KLW06].

## 4.4 Decoding based on Non-Binary Graphs

In many cases normal graphs include non-binary variables and generalized constraint nodes. Typical examples of such non-binary graphs are trellis representations of codes and graph representations of turbo codes. Non-binary variables necessitate non-binary message representations in the decoder. Depending on the implementation of the decoder one message representation or the other may be preferable. The messages form message vectors which are processed in generalized constraint node processors. Examples of decoding algorithms based on non-binary code graphs include the well known BCJR algorithm [BCJR74], the tailbiting APP decoding algorithm [AH98] for tailbiting codes and iterative decoding of turbo codes [BGT93]. All of them are covered in this section. We then summarize a commonly used sliding window technique which is later adapted to analog decoding. Iterative decoding of turbo codes is of particular importance since channel capacity may be closely approached despite the presence of cycles in the code graph.

### 4.4.1 Messages Representations for Non-Binary Random Variables

Message representations for non-binary random variables are obtained by a straightforward generalization of the binary message representations in Section 4.3.1. In the following we consider a discrete random variable $X$ with $x \in \mathcal{X}$ and $\mathcal{X} = \{0, \ldots, J-1\}$, where the total number of possible outcomes is $J$. The probability that random variable $X$ takes on values $x$ is denoted by $P_X(x)$ with $\sum_{x\in\mathcal{X}} P_X(x) = 1$. The log-likelihood of probability $P_X(x)$ is defined as $\ln(P_X(x)) = l_x(X)$. A log-likelihood ratio of the discrete random variable $X$ can be expressed by using two arbitrary outcomes $x$ and $x'$ [Ber00]

$$L_{x,x'}(X) = l_x(X) - l_{x'}(X) = \ln \frac{P_X(x)}{P_X(x')}, \tag{4.20}$$

where $L_{x,x'}(X) = -L_{x',x}(X)$, $L_{x,x}(X) = 0$ and $L_{x,x'}(X) = L_{0,x'}(X) - L_{0,x}(X)$. The log-likelihood ratio $L_{x,x'}(X)$ is then referred to as the L-value of random variable $X$ based on the outcomes $x$ and $x'$. The random variable $X$ is fully described by $J$ distinct probabilities or log-likelihoods as illustrated in Fig. 4.4. This information can be represented as vector of probabilities

$$(\, P_X(0), P_X(1), \ldots, P_X(J-1)\,), \tag{4.21}$$

or as vector of log-likelihoods

$$(\, l_0(X), l_1(X), l_2(X), \ldots, l_{J-1}(X)\,). \tag{4.22}$$

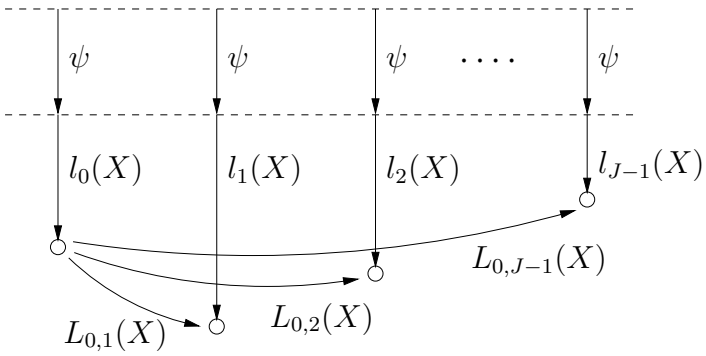



**Figure 4.4**: Illustration of log-likelihoods and L-values for non-binary variables.

Alternatively, we can also use $(J-1)$ L-values as depicted in Fig. 4.4 with $x=0$ as common reference. We then obtain the vector of L-values

$$( L_{0,1}(X), L_{0,2}(X), \ldots, L_{0,J-1}(X) ) . \tag{4.23}$$

In case of scaled probabilities according to $\kappa\, P_X(x)$ with $\sum_{x\in\mathcal{X}} \kappa\, P_X(x) = \kappa$ we obtain

$$\ln(\kappa\, P_X(x)) = \ln(\kappa) + l_x(X) = \psi + l_x(X), \tag{4.24}$$

where the additive constant $\psi$ corresponds to the natural logarithm of the scaling factor $\kappa$. Note that the L-values in (4.23) are not affected by such a scaling factor.

The probability of outcome $x$ can be calculated based on the log-likelihood ratios according to

$$P_X(x) = \frac{e^{-L_{x',x}(X)}}{\sum_{\nu=0}^{J-1} e^{-L_{x',\nu}(X)}} = A(X)\, e^{-L_{x',x}(X)}, \tag{4.25}$$

where the outcome $x'$ is used as common reference to be freely chosen within the set of possible outcomes, e.g., $x'=0$ in Fig. 4.4. Note that the scaling factor $A(X)$ in (4.25) cancels out when we convert the probabilities back to L-values.

### 4.4.2 Trellis Decoding

In this section we focus on the symbol-by-symbol APP decoding algorithm based on the trellis representation of codes [BCJR74]. This decoding algorithm is also known as BCJR algorithm according to the initials of the authors in [BCJR74]. The APP decoding algorithm can be viewed as a special case of message passing decoding on a cycle-free code graph where the messages and the constraint nodes are in general non-binary. The normal graph of a terminated convolutional code is depicted in Fig. 4.5 a) together with the corresponding trellis representation for the example of a memory $m=2$ code in Fig. 4.5 b). Note that all constraint nodes in the normal graph have degree three. One node processor of the associated decoder is exposed in Fig. 4.5 a) illustrating the bidirectional message exchange on the code graph. The decoder output is given in terms of log-likelihood ratios [HOP96] as

$$L(\hat{U}_k) = \ln \frac{P(u_k=0|\boldsymbol{y})}{P(u_k=1|\boldsymbol{y})} = \ln \frac{\sum\limits_{(s,s'),u_k=0} p(s,s',\boldsymbol{y})}{\sum\limits_{(s,s'),u_k=1} p(s,s',\boldsymbol{y})}, \tag{4.26}$$

where $\boldsymbol{y}$ denotes the sequence of received values and $s$ and $s'$ are possible realizations of state variables $S$ and $S'$ at time $k$ and $k+1$, respectively. For a memoryless transmission channel we

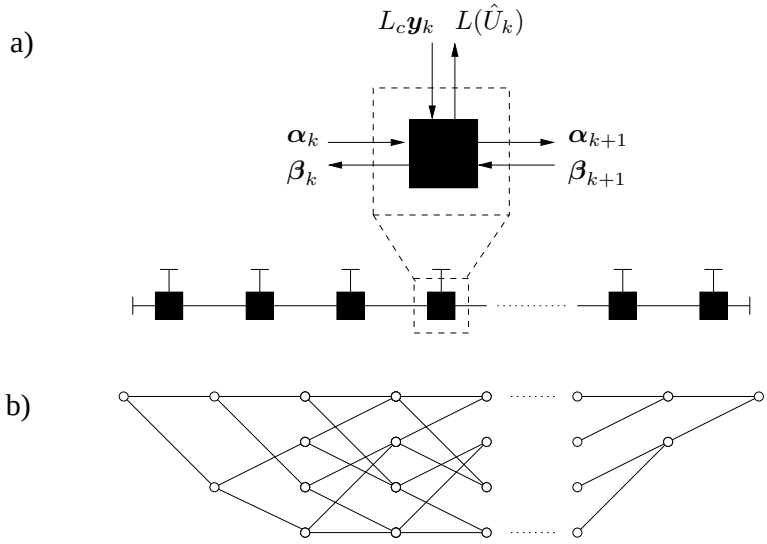



**Figure 4.5**: Normal graph of a terminated convolutional code with an exposed message flow for generalized constraint node $k$ in a) and the equivalent trellis representation for the example of a memory $m = 2$ code in b).

can define three independent probabilities [BCJR74], [HOP96]

$$\begin{aligned} \alpha_k(s) &= p(s, \boldsymbol{y}_{t<k}), && (4.27)\\ \gamma_k(s, s') &= p(s', \boldsymbol{y}_k|s) = P(s'|s)p(\boldsymbol{y}_k|s, s'), && (4.28)\\ \beta_{k+1}(s') &= p(\boldsymbol{y}_{t>k}|s'). && (4.29) \end{aligned}$$

This allows the substitution of $p(s, s', \boldsymbol{y})$ in the numerator and denominator of (4.26) with

$$p(s, s', \boldsymbol{y}) = \alpha_k(s)\gamma_k(s, s')\beta_{k+1}(s'). \tag{4.30}$$

The transition from state $s$ to state $s'$ in (4.28) is determined by

$$\gamma_k(s, s') = \prod_{\nu=1}^{n} \gamma_k^{(\nu)}(s, s') \tag{4.31}$$

with

$$\gamma_k^{(\nu)}(s, s') = \begin{cases} \dfrac{1}{1 + e^{-L_c y_k^{(\nu)}}} P(x_k^{(\nu)} = 0) & \text{if } x_k^{(\nu)} = 0, \\ \dfrac{e^{-L_c y_k^{(\nu)}}}{1 + e^{-L_c y_k^{(\nu)}}} P(x_k^{(\nu)} = 1) & \text{if } x_k^{(\nu)} = 1. \end{cases} \tag{4.32}$$

Here, $L_c y_k^{(\nu)}$ denotes the L-value associated with the $\nu$-th code bit in the $k$-th trellis section which is obtained from the channel. The probabilities $P(x_k^{(\nu)} = 0)$ and $P(x_k^{(\nu)} = 1)$ in (4.32) represent additional a priori information which may be available for the corresponding bit position. The $\alpha$ values in (4.27) are calculated in a forward recursion starting from the beginning of the trellis with

$$\alpha_{k+1}(s') = \sum_s \alpha_k(s)\gamma_k(s, s'), \tag{4.33}$$

while the $\beta$ values in (4.29) are calculated in a backward recursion starting from the end of the trellis with

$$\beta_k(s) = \sum_{s'} \gamma_k(s, s')\beta_{k+1}(s'). \tag{4.34}$$

The algorithm is therefore also referred to as forward-backward algorithm. In the following we assume that all paths in the code trellis start in the all-zero state and also end in the all-zero state. In this case, the $\alpha$ and the $\beta$ recursions are initialized with $\alpha_0(0) = 1$ and $\beta_{end}(0) = 1$, respectively. With (4.26) and (4.30) we obtain for the a posteriori decoder output for the $\nu$-th bit position in trellis section $k$

$$L(\hat{X}_k^{(\nu)}) = \ln \frac{\sum\limits_{(s,s'),x_k^{(\nu)}=0} \alpha_k(s)\gamma_k(s,s')\beta_{k+1}(s')}{\sum\limits_{(s,s'),x_k^{(\nu)}=1} \alpha_k(s)\gamma_k(s,s')\beta_{k+1}(s')}. \tag{4.35}$$

The expressions for the a posteriori decoder output in (4.35) can be split up into three independent terms [HOP96]

$$L(\hat{X}_k^{(\nu)}) = L_c y_k^{(\nu)} + L_a(X_k^{(\nu)}) + L_e(X_k^{(\nu)}), \tag{4.36}$$

where $L_c y_k^{(\nu)}$ represents the L-value obtained from the channel and $L_a(X_k^{(\nu)})$ as the optional a priori information. The third term $L_e(X_k^{(\nu)})$ reflects the extrinsic output of the decoder which is gathered from the information available about the other bit positions in the code word. In case of systematic codes the decoder output $L(\hat{U}_k)$ for the $k$-th information bit is determined by the corresponding code bit. When the information bit $U_k$ appears in state $S'$ of the $k$-th trellis section the expression in (4.35) simplifies to

$$L(\hat{U}_k) = \ln \frac{\sum\limits_{s',u_k=0} \alpha_{k+1}(s')\beta_{k+1}(s')}{\sum\limits_{s',u_k=1} \alpha_{k+1}(s')\beta_{k+1}(s')}. \tag{4.37}$$

This output calculation for example applies to convolutional codes with feedforward encoder.

In order to establish the link to message passing decoding we define the row vector $\boldsymbol{\alpha}_k$ and the column vector $\boldsymbol{\beta}_k$ according to

$$\boldsymbol{\alpha}_k = (\ \alpha_k(0), \ldots, \alpha_k(2^m - 1)\ ) \tag{4.38}$$

and

$$\boldsymbol{\beta}_k = (\ \beta_k(0), \ldots, \beta_k(2^m - 1)\ )^T, \tag{4.39}$$

respectively. Both vectors have a total number of $2^m$ elements and represent the probabilities of trellis state $S$ in the forward and backward recursion. Furthermore, we define a $2^m \times 2^m$ transition matrix $\boldsymbol{\Gamma}_k$ with elements $\gamma_k(s, s')$. Transitions which are not present in the trellis are represented with a zero entry in $\boldsymbol{\Gamma}_k$. The forward and backward recursion of the decoding algorithm can then be defined in vector matrix notation as

$$\boldsymbol{\alpha}_{k+1} = \boldsymbol{\alpha}_k \boldsymbol{\Gamma}_k \tag{4.40}$$

and

$$\boldsymbol{\beta}_k = \boldsymbol{\Gamma}_k \boldsymbol{\beta}_{k+1}, \tag{4.41}$$

respectively. This formulation allows the interpretation of the APP decoding algorithm as message passing decoding based on the normal graph in Fig. 4.5 a). The vectors in (4.38) and (4.39) represent the messages while (4.40) and (4.41) define the local computations in the node processors. The messages are scheduled in a way that the output of a node processor is calculated as soon as the corresponding inputs become available. With the appropriate initialization of the message vectors at the beginning end the end of the terminated trellis this leads to a propagation of the messages in forward and backward direction along the code graph. A node processor

can calculate the decoder output according to (4.35) or (4.37) as soon as the messages from the forward and backward recursions are available.

A straightforward implementation of the algorithm in the digital domain uses only a single node processor for each recursion. One starts at the beginning of the trellis and processes the information in the forward recursion and stores all intermediate $\boldsymbol{\alpha}$ vectors until the end of the trellis is reached. As soon as the channel values of the overall code block have been received the second processor can start from the end of the trellis and calculate the $\boldsymbol{\beta}$ vectors in the backward recursion. The results of this recursion are then combined with the stored vectors of the forward recursion and the channel values in order to calculate the decoder output. This procedure guarantees optimal decoder performance, but introduces a considerable amount of decoding delay, in particular for codes with large block lengths. Furthermore, the sequence of all channel values (or $\gamma$ values equivalently) and $\boldsymbol{\alpha}$ vectors need to be stored along the trellis, which uses up lots of memory. It is therefore common practice to utilize a sliding window decoder as described in the next section.

### 4.4.3 Generalized Sliding Window Decoding

A digital decoder is commonly implemented using a sliding window technique [DGM93], [BDMP96], [Vit98] (and the references therein) in order to reduce decoding delay and storage requirements. This approach can be thought of as message passing decoding on a fragment of the code graph rather than the overall code graph. For the formulation of the generalized sliding window decoding algorithm we assume that such a fragment consists of $W = L + D + L$ consecutive trellis sections as shown in Fig. 4.6. Typically, $W$ spans significantly less code bits

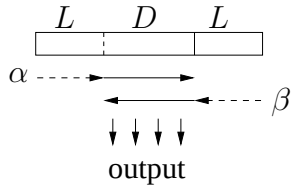



**Figure 4.6**: Generalized sliding window decoder with a window size of $W = D + 2L$.

than the overall code trellis. The $W$ trellis sections are referred to in the following as the decoding window. The APP decoding algorithm from Section 4.4.2 is now applied to such decoding windows. In general, there is no information available about the distribution of the $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ vectors at the beginning and the end of the decoding window. Here, sliding window decoding relies on the fact that the recursions approximate the distributions of the $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ vectors in the APP decoder after a sufficiently large number of trellis sections independent of the starting distribution. This number of trellis sections is referred to as stabilization length $L$. As a rule of thumb, the stabilization length is typically five to six times the encoder memory. We always assume that the sliding window decoder starts with a uniform distribution of state probabilities unless there is some other information available, e.g., at the beginning and the end of a conventional trellis. In case the stabilization length $L$ is required for both recursions we can decode $D = W - 2L$ trellis sections within each decoding window. Typically, the decoding window slides along the code trellis as the channel values are received. There is also the possibility of decoding more than one window in parallel since all decoding windows are independent of each other. The speed of the decoder then scales linearly with the number of windows processed in parallel. The generalized sliding window decoder allows a significant reduction of both decoding delay and storage requirements at the expense of a computational overhead introduced by the stabilization length $L$. We will see later that the generalized sliding window

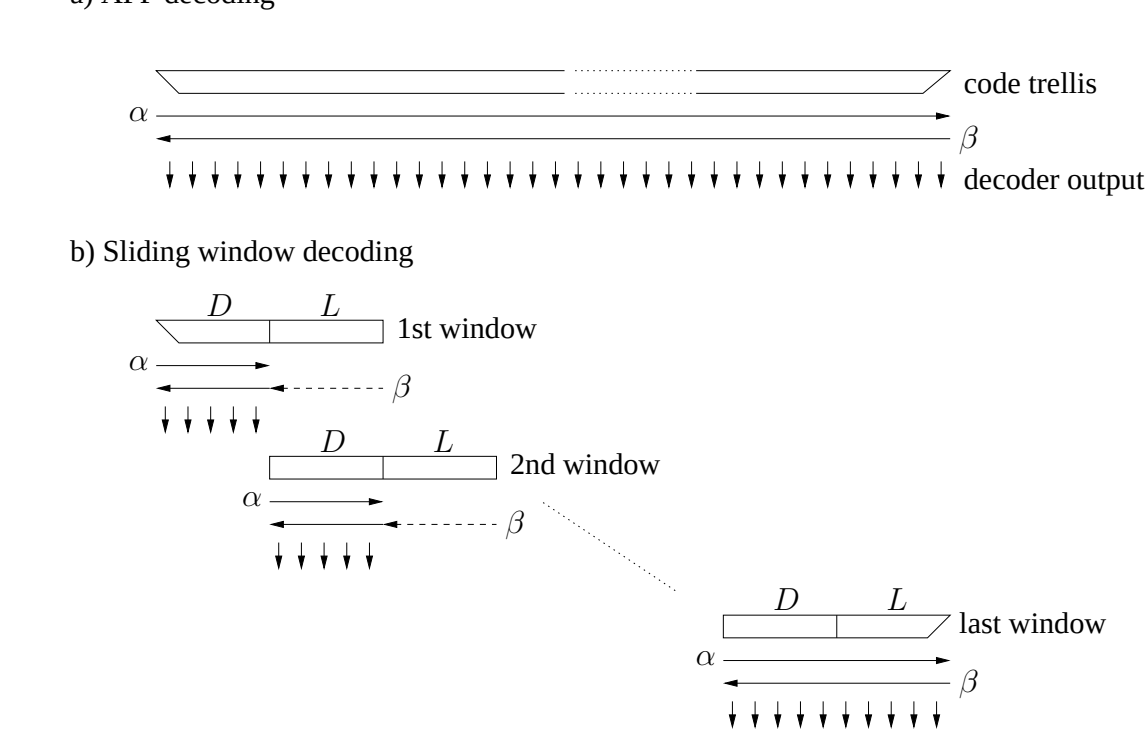



**Figure 4.7**: APP decoding of a terminated convolutional code in a) and corresponding sliding window decoding with $W = D + L$ in b).

decoder, although being suboptimal, in many cases closely approaches the performance of the APP decoder.

A similar technique was utilized earlier for high-speed Viterbi decoding [FM91] and the equalization of inter-symbol interference (ISI) channels [AF70], [EPG94].

In the following we investigate two instances of generalized sliding window decoding. The first realization of this technique applies to terminated convolutional codes. Conventional APP decoding and sliding window decoding of such a code is illustrated in Fig. 4.7 a) and Fig. 4.7 b), respectively. The decoding window of size $W = D+L$ with $D = L$ starts at the beginning of the code trellis and then slides along the code trellis. The $\alpha$ recursion in the first decoding window is initialized according to the APP decoder in Section 4.4.2 so that there is no stabilization length $L$ required. The $\beta$ recursion is initialized with a uniform distribution of state probabilities as described in the above. Note that this recursion can start much earlier than the $\beta$ recursion in the APP decoder, compare Fig. 4.7 a). Further note that the $\alpha$ recursion is only performed for $D$ trellis sections while the $\beta$ recursion runs over $L + D$ trellis sections. After the first $D = W - L$ trellis sections are decoded the decoding window is shifted by $D$ trellis sections to the right in order to decode the next $D$ trellis sections. The $\alpha$ recursion in the second decoding window is now initialized with the result of the $\alpha$ recursion in the first decoding window and the $\beta$ recursion proceeds as described for the first decoding window. The last decoding window benefits from the know starting distribution for the $\beta$ recursion at the end of the code trellis so that up to $D + L$ trellis sections can be decoded. In order to achieve a more continuous decoder output there can be one processing unit for the forward recursion and two processing units for $\beta$ recursion. While one unit calculates $\boldsymbol{\beta}$ vectors used for the decoder output, the other unit already builds up reliable $\boldsymbol{\beta}$ vectors in the next decoding window. When the last output of one decoding window is calculated the decoder can then continue with output calculations in the

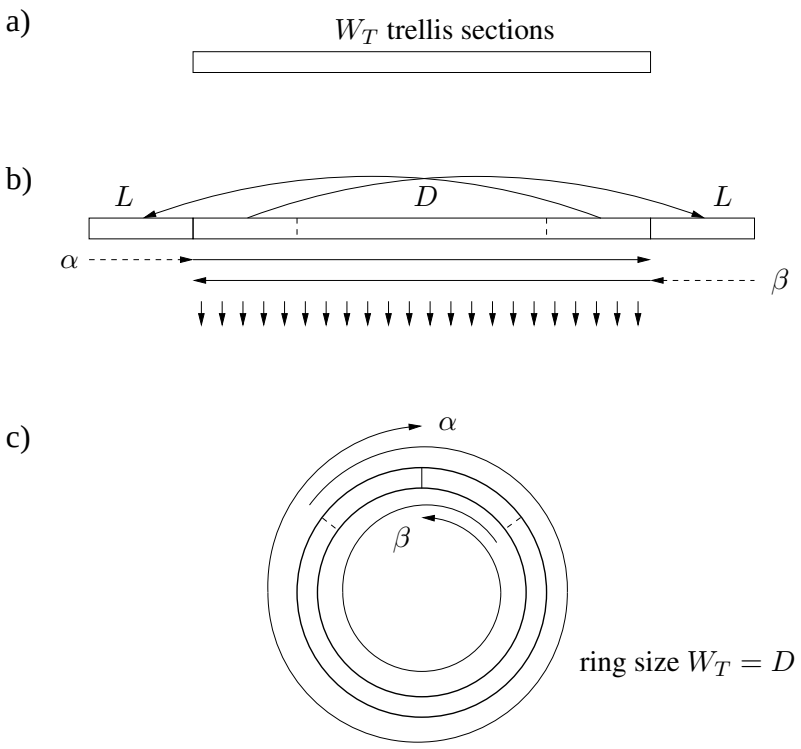



**Figure 4.8**: a) Unwrapped trellis of a tailbiting code. b) Decoding of a tailbiting code as a special case of the generalized sliding window decoding algorithm. c) Decoding of a tailbiting code with the wrap around algorithm.

next decoding window.

The second realization of generalized sliding window decoding applies to tailbiting convolutional codes. Optimal decoding of tailbiting convolutional codes requires to run the APP decoding algorithm from Section 4.4.2 for each of the $2^m$ possible starting states in the trellis. This is because the state at the beginning and the end of the tailbiting trellis is not necessarily the first, i.e., the all-zero state. Instead, any state is possible with the only constraint that the first and the last state in the trellis is the same for the set of all code words. This implies that the optimal decoder for a tailbiting convolutional code is roughly by a factor of $2^m$ more complex than the decoder for a terminated convolutional code. The key challenge in decoding tailbiting convolutional codes is the unknown starting distribution of the recursions at the beginning and the end of the trellis. This problem can efficiently be solved with the generalized sliding window decoding algorithm from above. For this, the parameter $D$ is chosen according to the number of trellis sections in the tailbiting ring. The stabilization of the forward and backward recursions then occurs on the last and the first $L$ trellis sections of the tailbiting trellis, respectively. The unwrapped trellis of a tailbiting convolutional code with $W_T$ trellis sections is shown in Fig. 4.8 a). Fig. 4.8 b) illustrates how the decoding window of size $W = L + W_T + L$ derives from Fig. 4.8 a). We then start the forward and the backward recursions with a uniform distribution of state probabilities and calculate the decoder output for the $D = W_T$ trellis sections of the tailbiting convolutional code.

The problem of finding the appropriate starting state in the tailbiting trellis of the code can also be formulated as eigenvalue problem

$$\boldsymbol{\alpha}_0 P(\boldsymbol{y}) = \boldsymbol{\alpha}_L = \boldsymbol{\alpha}_0 \boldsymbol{\Gamma}_1 \dots \boldsymbol{\Gamma}_{W_T}, \tag{4.42}$$

where the starting distribution $\boldsymbol{\alpha}_0$ corresponds to the normalized left eigenvector of the matrix $\boldsymbol{\Gamma}_1 \dots \boldsymbol{\Gamma}_{W_T}/P(\boldsymbol{y})$ [AH98]. Similarly, the backward recursion can be initialized with the right

eigenvector of the matrix. The calculation of these eigenvalues can be avoided due to the fact that the repeated multiplication of an arbitrary starting distribution $\boldsymbol{\alpha}_0$ with a positive matrix converges to the principal eigenvalue of that matrix [Dei91]. This means that we reasonably well approximate these eigenvalues and thus the desired starting distributions for the recursions when we perform the forward and backward recursions several times around the tailbiting trellis. Typically, we do not have to perform several recursions around the tailbiting trellis unless the block length is very short with respect to the encoder memory, i.e., $W_T < L$. Fig. 4.8 c) depicts the generalized sliding window decoder based on the tailbiting trellis of the code. The ring structure of the code naturally connects the beginning and the end of the decoding window and therefore provides the required stabilization length at the beginning and the end of the window automatically. Depending on the stabilization length $L$ and the block length $W_T$ the recursions wrap more or less around the decoder ring. The algorithm is therefore also referred to as the wrap-around decoding algorithm.

An analog decoder implements such a ring for both the forward and the backward recursion. In this case, neither the stabilization length is limited to $L$ trellis sections, nor the recursions start with a uniform distribution of the $\alpha$ and the $\beta$ values. Instead, the signals propagate freely around the tailbiting trellis until a stable state is reached. The analog tailbiting convolutional decoder is covered in more detail in Section 5.2.3.

Note that the decoding window in Fig. 4.8 b) can also be split up into several decoding windows.

### 4.4.4 Iterative Decoding of Turbo Codes

The parallel concatenation of block or convolutional codes is a particularly interesting example of codes based on non-binary graphs. Such a code construction was first devised by Berrou, Glavieux and Thitimajshima for the case of two parallel concatenated convolutional codes together with a very powerful iterative decoding algorithm [BGT93]. Due to their unprecedented good error correcting performance these codes were termed turbo codes. The normal graph of such a code concatenation is based on the normal graphs of the two component codes which are connected to each other through equality constraint nodes and an interleaver network $\Pi$ as shown in the lower part of Fig. 4.9. The scheduling of the messages on this code graph is summarized below.

The turbo decoder is divided into the set of the upper and lower node processors which form the decoders for the two component codes. The bidirectional message exchange within the first component decoder and between the two component decoders is exposed in the upper part of Fig. 4.9 for the example of two rate $R = 1/2$ convolutional codes. The input values $L_c y_k^{(1)}$ and $L_c y_k^{(2)}$ represent the channel values for the $k$-th information bit $u_k = x_k^{(1)}$ and the corresponding parity bit $x_k^{(2)}$ associated with the $k$-th node processor (trellis section) of the first component decoder. The node processors of the component codes are in general non-binary and involve the branch metric computation, the calculations associated with the forward and backward recursions and the calculation of the extrinsic output as outlined in Section 4.4.2. At the beginning of the decoding process a new block of channel values $L_c\boldsymbol{y}$ is received and all decoder internal L-values are reset to zero, i.e.

$$L_{e,\xi}(U_k) = 0, \quad \forall k \text{ and } \xi \in \{1, 2\}. \tag{4.43}$$

One iteration of the iterative decoding process is then divided into the following steps. In the first half of the iteration the equality node processors provide

$$L_{in,1}(U_k) = L_c y_k^{(1)} + L_{e,2}(U_k), \quad \forall k, \tag{4.44}$$

to the input of the first component decoder. This includes the channel information $L_c y_k^{(1)}$ about the systematic bit and the extrinsic output $L_{e,2}(U_k)$ of the second component decoder, which acts

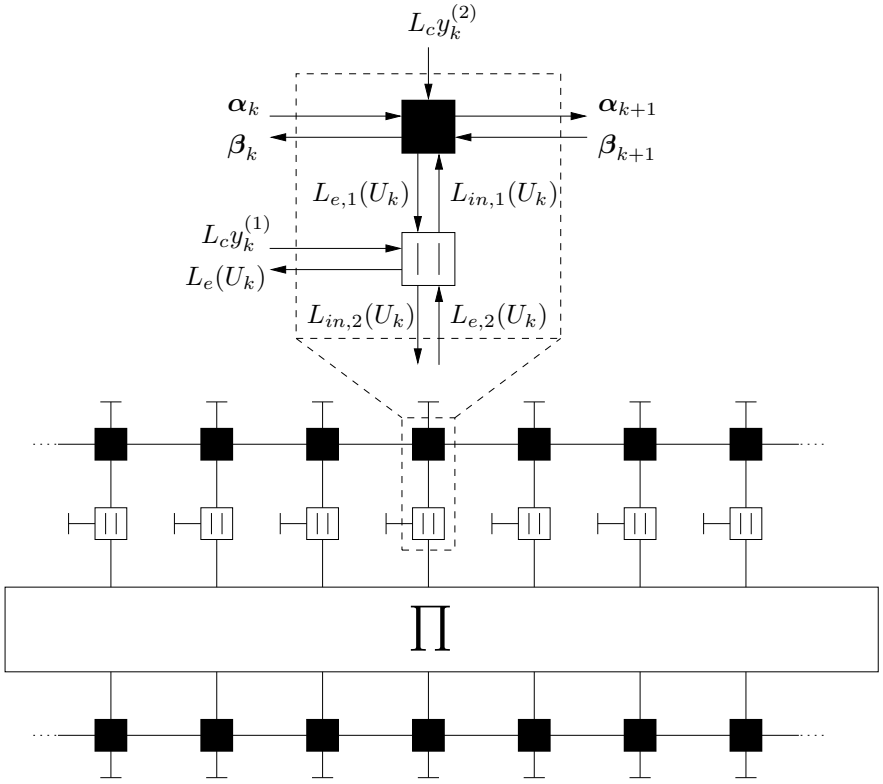



**Figure 4.9**: Normal graph of a turbo code which consists of a parallel concatenation of two rate $R = 1/2$ convolutional codes.

as a priori information for the first component decoder. The channel values $L_c y_k^{(2)}$ associated with the first set of parity bits are applied concurrently. The first component decoder then calculates its extrinsic output $L_{e,1}(U_k)$ according to the scheduling of the messages in Section 4.4.2. This extrinsic output is passed back to the equality node processors. The equality node processors then provide

$$L_{in,2}(U_k) = L_c y_k^{(1)} + L_{e,1}(U_k), \quad \forall k, \tag{4.45}$$

to the interleaver. This includes the channel information $L_c y_k^{(1)}$ about the systematic bit and the extrinsic output $L_{e,1}(U_k)$ of the first component decoder, which acts as a priori information for the second component decoder. In the second half of the iteration the interleaved version of $L_{in,2}(U_k)$ is applied to the input of the second component decoder. The channel values associated with the second set of parity bits originating from the second component encoder are applied concurrently. The second component decoder then calculates its extrinsic output $L_{e,2}(U_k)$ according to the scheduling of the messages in Section 4.4.2. This extrinsic output is then de-interleaved and passed back to the equality node processors. This step completes one iteration of the turbo decoder. The decoding process than continues for a fixed number of iterations, or, until a certain stopping criteria is fulfilled, see, e.g., [Rob94]. After the last decoder iteration the equality node processors provide the extrinsic output of the overall turbo decoder, i.e.,

$$L_e(U_k) = L_{e,1}(U_k) + L_{e,2}(U_k), \quad \forall k, \tag{4.46}$$

or, alternatively, the overall decoder output

$$L(\hat{U}_k) = L_c y_k^{(1)} + L_{e,1}(U_k) + L_{e,2}(U_k), \quad \forall k. \tag{4.47}$$

## 4.5 Extrinsic Information Transfer Charts

The extrinsic information transfer (EXIT) chart was introduced by ten Brink in 1999 [tB99]. EXIT charts are nowadays a commonly used tool for the analysis of iterative decoding and the design of new codes. They rely on the mutual information as introduced in Section 2.5. One of the key advantages is that the convergence of iterative decoding can be predicted solely by investigating the component decoders instead of the overall iterative decoder. This significantly simplifies the analysis and allows the selection of component codes with matching properties. Every component decoder is described by a characteristic curve in the EXIT chart. In case the received channel values are applied to the input of the component decoder this characteristic curve depends on the channel SNR. The distribution of the L-values obtained from the channel can be derived from the distribution of the matched filter output $y$ using the transformed random variable $y' = L_c y$ with the conditioned pdf

$$\begin{aligned} p_{y'}(L_c y|x=\pm 1) &= p_y(1/L_c\, y'|x=\pm 1)\frac{1}{L_c} \\ &= \frac{1}{\sqrt{4\pi L_c}}\, e^{-\frac{(y'\mp L_c)^2}{4L_c}} \\ &= \frac{1}{\sqrt{2\pi\sigma_{y'}^2}}\, e^{-\frac{(y'\mp m_{y'})^2}{2\sigma_{y'}^2}}. \end{aligned} \tag{4.48}$$

We obtain a Gaussian distribution with mean $m_{y'} = \pm L_c = 2/\sigma_n^2$ and variance $\sigma_{y'}^2 = 2L_c = 4/\sigma_n^2$. The conditioned pdf in (4.48) is symmetric, i.e., $p(-y'|x) = p(y'|-x)$, and for equally likely inputs with $P(x=+1) = P(x=-1)$ satisfies the consistency condition [RU01]

$$p(-y'|x) = e^{-y'x} p(y'|x). \tag{4.49}$$

Note that the consistency condition is fulfilled for a Gaussian distribution only if $\sigma_{y'}^2 = 2m_{y'}$. This implies that the L-values from the channel are fully characterized by the SNR, or, alternatively, by the mean or the variance of the distribution. Another possible characterization is the mutual information. For the symmetric (and consistent) pdf in (4.48) we obtain for the mutual information [TH02]

$$I(X;Y') = 1 - \int_{-\infty}^{\infty} p(y'|x=+1)\log_2(1+e^{-y'})\, dy' \tag{4.50}$$

$$= 1 - \mathrm{E}\{\log_2(1+e^{-y'})|x=+1\}. \tag{4.51}$$

When we replace the expectation $\mathrm{E}\{.\}$ with the time average over a sufficiently large number of L-values we obtain the approximation

$$I(X;Y') \approx 1 - \frac{1}{N_I}\sum_{i=1}^{N_I}\log_2(1+e^{-x_i y_i'}). \tag{4.52}$$

The simulation of one component decoder independent of the other requires that the a priori information is modeled appropriately. It is common practice in the EXIT chart analysis to assume that the L-values $L_a$ at the a priori input of the decoder are also distributed according to (4.48). We then obtain

$$L_a = \frac{\sigma_a^2}{2}x + \sigma_a, \tag{4.53}$$

with mean $m_a = \pm\sigma_a^2/2$ and variance $\sigma_a^2$ of a consistent Gaussian distribution. The mutual information at the a priori input can be expressed in terms of $\sigma_a$ as

$$I_A(\sigma_a) = I(X; L_a) = 1 - \int_{-\infty}^{\infty} \frac{1}{\sqrt{2\pi\sigma_a^2}} e^{-\frac{(L_a - \sigma_a^2/2)^2}{2\sigma_a^2}} \log_2(1 + e^{-L_a})\, dL_a, \tag{4.54}$$

which can be solved numerically. It is monotonically increasing in $\sigma_a$ so that there is a unique $\sigma_a$ for a given $I_A(\sigma_a)$.

The characteristic curve of a component decoder illustrates the mutual information $I_E = I(X; L_e)$ at the extrinsic output of the component decoder as a function of the mutual information $I_A(\sigma_a)$ at its a priori input. We now summarize the simulation of a characteristic curve for a given $E_b/N_0$. Randomly generated information bits are encoded and transmitted over an AWGN channel with

$$\sigma_n^2 = \frac{N_0}{2E_S} = \frac{N_0}{2RE_b}, \tag{4.55}$$

where $R$ refers to the code rate of the concatenated coding scheme. The mutual information at the input of the component decoder is chosen so that $0 \leq I_A(\sigma_a) \leq 1$. A selected $I_A(\sigma_a)$ value then determines $\sigma_a$ as described in the above. This allows the modeling of the L-values at the a priori input with a Gaussian (and consistent) distribution according to (4.53). After the calculation of the L-values at the extrinsic output of the component decoder we can determine the mutual information $I_E$ according to (4.52). The simulations are then repeated so that a sufficiently large number of L-values is available for the calculation of the mutual information.

The performance of an iterative decoder can be predicted based on the plot of the characteristic curves of the two component decoders. The mutual information $I_{E1}$ at the extrinsic output of the first component decoder represents the mutual information $I_{A2}$ at the a priori input of the second component decoder. The mutual information at the extrinsic output $I_{E2}$ of the second component decoder then forms the mutual information $I_{A1}$ at the a priori input of the first component decoder. The characteristic curves are therefore combined in a way that $I_{A1}$ and $I_{E2}$ are plotted on the x-axis while $I_{E1}$ and $I_{A2}$ are represented on the y-axis of the EXIT chart. The trajectory of the iterative decoder can then be determined graphically based on the plot of the two characteristic curves. The first component decoder starts with $I_{A1} = 0$ and provides its extrinsic output to the a priori input of the second component decoder, i.e., $I_{A2} = I_{E1}$. The second component decoder then feeds back its extrinsic output to the a priori input of the first component decoder, i.e., $I_{A1} = I_{E2}$, and a new iteration starts. This process then continues for a certain number of iterations. The zig-zag line in the EXIT chart which illustrates this iterative exchange of mutual information represents the estimated trajectory of the overall decoder. This estimation improves with increasing block length of the code. Depending on the SNR of the channel the characteristic curves of the two component decoders may intersect or not. In case the curves do not intersect and there is an open tunnel in the EXIT chart from the point $(I_{A1} = 0, I_{A2} = 0)$ to the point $(I_{E1} = 1, I_{E2} = 1)$ the zig-zag line may reach the upper right corner with increasing number of iterations. At the point $(I_{E1} = 1, I_{E2} = 1)$ the decoder has perfect knowledge about the transmitted bits, i.e., the bit error probability is zero.

More details on the EXIT chart analysis of serially concatenated codes and parallel concatenated codes can be found in [tB00] and [tB01], respectively.

## 4.6 Quantization of Soft Information

This section focuses on the quantization of soft information as it is necessary in all digital receivers. Here, the performance of a floating point implementation can be closely approached by a realization which uses only a finite number of quantization levels. The number of quantization levels in a digital decoder determines the computational complexity of the node processors

and the size of the memory in the decoder. The goal is therefore to use a minimum number of quantization levels while maintaining an acceptable decoder performance. An analog implementation theoretically works with an infinite number of quantization levels, i.e., the true soft values. However, device mismatch, temperature dependencies and other impairments may cause an effect similar to quantization. Sensitivity to quantization would thus also indicate a sensitivity to such impairments. Quantization of soft information is also particularly important since in most applications analog decoders need to interface to other digital components in the receiver. This implies that there are digital-to-analog (D/A) and analog-to-digital (A/D) converters at the input and the output of the analog decoder, respectively. The number of bits which need to be processed in the D/A and A/D converter directly impact area and power consumption of these converters. In many practical applications it may also be required to store soft information inside the decoder in order to facilitate sequential decoder architectures. An example of this is the sliding window turbo decoder architecture in Section 5.4.2 where the extrinsic output of the component decoders can optionally be stored in the digital domain. In this case the required number of bits for the internal A/D and D/A conversion is of particular importance since these converters need to operate at a multiple of the block rate. This is because one iteration requires the A/D conversion of all extrinsic outputs and the corresponding D/A conversion after interleaving or de-interleaving. The number of quantization levels is also important for analog decoders where the input from the channel or the extrinsic information is stored on analog memory elements, e.g., as voltage on a capacitor. Here, the digital resolution gives an indication for the required accuracy of the analog storage elements, e.g., the size of the capacitor.

In the following we summarize a nonlinear quantization technique which is based on the computational cutoff rate [Mas74], [JZ99]. The computational cutoff rate $R_0$ provides, in contrast to the Shannon limit, a more practical measure for the achievable information rate in a communication system with moderate decoder complexity [Fri95]. This quantization technique is optimal in the sense that it maximizes $R_0$ for all possible quantization schemes using an identical number of quantization levels. The computational cutoff rate for a discrete memoryless channel (DMC) with binary input $x$ and $q$-ary output $\bar{y}^{(\nu)}$, $\nu \in \{1, \ldots, Q\}$, is given by

$$R_0 = -\log_2 \left( \min_{P(x)} \left\{ \sum_{\nu=1}^{Q} \left( \sum_{x} \sqrt{P(\bar{y}^{(\nu)}|x)} P(x) \right)^2 \right\} \right), \tag{4.56}$$

where the minimization is carried out for all distributions $P(x)$ at the input [Mas74]. In our context, the output of the DMC represents the different quantization levels $\bar{y}^{(\nu)}$ while $P(\bar{y}^{(\nu)}|x)$ denotes the transition probability from input $x$ to quantization level $\bar{y}^{(\nu)}$. For equally likely input bits, i.e., $P(x = +1) = P(x = -1) = 0.5$, the expression in (4.56) is maximized and we obtain

$$R_0 = 1 - \log_2 \left( 1 + \sum_{\nu=1}^{Q} \sqrt{P(\bar{y}^{(\nu)}|x = +1) P(\bar{y}^{(\nu)}|x = -1)} \right). \tag{4.57}$$

In this case the computational cutoff rate only depends on the transition probabilities $P(\bar{y}^{(\nu)}|x)$ which are determined by the selection of the quantization thresholds $T_\nu$. Here, we assume that $T_\nu$ represents the threshold between the quantized outputs $\bar{y}^{(\nu)}$ and $\bar{y}^{(\nu+1)}$. The quantization is assumed to be optimal when the selected quantization thresholds maximize the computational cutoff rate and thus minimize the sum in (4.57). A necessary condition for this can be formulated according to [Mas74], [JZ99] as

$$L(y = T_\nu|X) = \frac{1}{2} \left( L(\bar{y}^{(\nu)}|X) + L(\bar{y}^{(\nu+1)}|X) \right), \tag{4.58}$$

with

$$L(y = T_\nu|X) = \ln \frac{p(y = T_\nu|x = +1)}{p(y = T_\nu|x = -1)} \tag{4.59}$$

and

$$L(\bar{y}^{(\nu)}|X) = \ln \frac{P(\bar{y}^{(\nu)}|x = +1)}{P(\bar{y}^{(\nu)}|x = -1)}. \tag{4.60}$$

The quantization thresholds $T_\nu$ can be found with a simple algorithm [Mas74], [JZ99]. Start with an arbitrarily selected $T_1$ which determines $L(y = T_1|X)$ and $L(\bar{y}^{(1)}|X)$. Choose $T_2$ so that $L(\bar{y}^{(2)}|X)$ satisfies (4.58). Then choose $T_3$ so that $L(\bar{y}^{(3)}|X)$ satisfies (4.58) and so on. In case a $T_{Q-1}$ is found which also satisfies (4.58) the approach was successful. Otherwise, the procedure needs to be repeated for a different $T_1$.

The above algorithm determines the quantization thresholds $T_\nu$ for the matched filter output $y$ depending on the distribution of the channel values, i.e., the channel SNR. Table 4.1 lists the optimal quantization thresholds together with the corresponding quantized L-values $L(\bar{y}^{(\nu)}|X)$ for the example of $E_S/N_0 = 0$ dB. The corresponding values for the computational cutoff rate are also added in Table 4.1. Alternatively, the quantization thresholds can also be calculated based on the distribution of L-values $p_{y'}(L_c y|x)$ in (4.48). In this case the quantization thresholds in Table 4.1 need to be scaled with $L_c$ while the quantization levels remain unchanged.

The channel values at the input of the decoder can directly be quantized according to Table 4.1. However, in Section 5.4.2 we cover a sliding window turbo decoder which necessitates the storage of the extrinsic output of the component decoders. In this case, the information can be stored on either analog or digital memory elements. The latter immediately raises the question about the appropriate quantization of the extrinsic L-values since the distribution of the extrinsic values differs from the distribution of the channel values. We demonstrate in Section 5.4.2 that an excellent decoder performance can be achieved when the scaled extrinsic output $L_e(X)/L_c$, with $L_c = 4E_S/N_0$, is quantized according to Table 4.1. This observation is also supported by the fact that a precise estimation of the channel SNR is not required for a good decoder performance.

**Table 4.1**: Quantization thresholds $T_\nu$ for the matched filter output $y$ and the corresponding quantized L-values $L(\bar{y}^{(\nu)}|X)$ for $E_S/N_0 = 0$ dB.

| $Q$ | $R_0$ | $T_\nu$ | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | $\nu=1$ | $\nu=2$ | $\nu=3$ | $\nu=4$ | $\nu=5$ | $\nu=6$ | $\nu=7$ | $\nu=8$ | $\nu=9$ | $\nu=10$ | $\nu=11$ | $\nu=12$ | $\nu=13$ | $\nu=14$ | $\nu=15$ |
| 2 | 0.378 | 0 | | | | | | | | | | | | | | |
| 4 | 0.498 | -0.73 | 0 | 0.73 | | | | | | | | | | | | |
| 8 | 0.534 | -1.27 | -0.76 | -0.36 | 0 | 0.36 | 0.76 | 1.27 | | | | | | | | |
| 16 | 0.543 | -1.72 | -1.32 | -1.02 | -0.78 | -0.57 | -0.37 | -0.18 | 0 | 0.18 | 0.37 | 0.57 | 0.78 | 1.02 | 1.32 | 1.72 |
| $\infty$ | 0.548 | | | | | | | | | | | | | | | |

| $Q$ | $L(\bar{y}^{(\nu)}\|X)$ | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $\nu=1$ | $\nu=2$ | $\nu=3$ | $\nu=4$ | $\nu=5$ | $\nu=6$ | $\nu=7$ | $\nu=8$ | $\nu=9$ | $\nu=10$ | $\nu=11$ | $\nu=12$ | $\nu=13$ | $\nu=14$ | $\nu=15$ | $\nu=16$ |
| 2 | -2.46 | 2.46 | | | | | | | | | | | | | | |
| 4 | -4.49 | -1.34 | 1.34 | 4.49 | | | | | | | | | | | | |
| 8 | -6.26 | -3.88 | -2.18 | -0.70 | 0.70 | 2.18 | 3.88 | 6.26 | | | | | | | | |
| 16 | -7.85 | -5.90 | -4.60 | -3.56 | -2.67 | -1.85 | -1.09 | -0.36 | 0.36 | 1.09 | 1.85 | 2.67 | 3.56 | 4.60 | 5.90 | 7.85 |

# 5

# *Analog Decoding*

FEC decoders are traditionally implemented using digital signal processors (DSPs), field programmable gate arrays (FPGAs) or digital application specific integrated circuits (ASICs). Digital decoder implementations operate on discrete messages in discrete time and require the distribution of clock signals within the decoder. In order to achieve a good performance, the decoder needs to process soft information. This requires that the channel values at the decoder input and the values inside the decoder are represented by a sufficiently large number of quantization levels. Many applications require that the decoder also provides soft information at its output. This soft output also plays an essential role when the decoder is used as a component decoder for state-of-the-art LDPC codes or turbo codes. Here, a significant number of iterations are necessary for decoding a single code word, during which the extrinsic output from one component decoder is fed back as a priori information to the input of the other component decoder.

However, looking at the basic concept of such a soft-in/soft-out decoder, it seems to be far more obvious and straightforward to use an analog signal processor instead of a digital one. An analog decoder naturally accepts and delivers soft values represented by voltages and currents in analog transistor circuits without any quantization loss. Here, decoding takes place in a completely unsynchronized and time-continuous fashion without the need for any internal clock signals. Compared to conventional iterative decoding, the iterations vanish and the extrinsic information is exchanged in a time-continuous fashion. Consequently, new information is passed on to neighboring node processors as soon as it becomes available in one part of the decoder. Overall decoder performance may thus be improved. This is in clear contrast to digital decoding where information is first calculated according to the decoding rule and then exchanged in one full step. The time- and value-continuous analog behavior could be modeled in the digital domain by using only small message increments instead of the calculated messages. Both the messages and the time axis would then be represented with a finer resolution. This would clearly increase the complexity of such a digital decoder implementation. Analog signal processing in the decoder naturally achieves a time- and value-continuous message exchange and thus captures potential performance gains at no extra cost. After a certain time, the voltages and currents in the analog network settle and the decoding result is available at the output. The settling speed is not only determined by the parasitics in the decoding network and the temperature, but also by the channel SNR and the configuration of channel values for a particular decoding scenario. An increasing channel SNR leads to a smaller probability for channel errors and thus speeds up the decoding process in general. However, certain decoding situations may still require a significantly longer amount of time. This effect needs to be carefully considered when analog decoders are simulated and in particular when the speed of these networks is being estimated.

Given the time-continuous characteristic of analog decoders we find that decoder performance always degrades gracefully when there is not enough time for the settling of the network.

The time-continuous exchange of information inside the decoder raises questions about the performance of an analog decoder compared to a digital decoder implementation. This question is particularly interesting when it comes to an analog implementation of an iterative decoding algorithm where the underlying code graph typically involves cycles. Here, the equivalence of an analog and a digital decoder is in general impossible to prove analytically. In most cases it will therefore only be possible to evaluate decoder performance by other means such as BER simulations or EXIT chart analysis.

This leads us to one of the greatest challenges, the appropriate simulation of analog decoders on digital (and in particular time-discrete) computer systems. We therefore pay particular attention to the simulation of analog decoders, as covered in Section 5.1. We will introduce a new unified design and simulation environment which allows the utilization of different time-continuous, time-discrete and circuit-level simulation models. Different analog decoding networks will be presented in the course of this chapter. All analog decoders are based on (transformed) normal graphs where the degree of the nodes is three. After the investigation of some basic analog decoding networks in Section 5.2, we will introduce a new analog sliding window technique which operates on only a small sub-graph of the overall code. Different architectures of analog decoders for state-of-the-art turbo codes and LDPC codes are covered in Section 5.4. This chapter concludes with comments on the possible equivalence between analog and digital decoding in Section 5.5.

## 5.1 Simulation of Analog Decoders

The simulation of analog decoders requires significantly more attention than the simulation of digital decoders. In the analog domain it is particularly important to capture the dynamics of the time-continuous message exchange within the decoding network appropriately. It directly impacts the quality of the soft output and hence the BER performance of the decoder. This dynamic behavior is mainly determined by the structure of the directed normal graph on which the analog decoder is based, the design and placement of the node processors on the chip and the routing of the interconnects. This introduces parasitic resistor and capacitor values in the decoding network which not only depend on the size of the selected transistors but also on the physical layout of the decoder. This includes the length and width of the interconnects, the selected layer of metal and the number of vias required in order to connect from one layer of metal to another. These parasitics cause processing and propagation delays which clearly limit the potential speed of the analog decoding network.[1] The most accurate simulations are thus achieved after the parasitics are extracted from the physical layout of the decoder chip and provided as input to the circuit-level simulator for post-layout simulations. Clearly, this adds additional complexity and thus further increases simulation time. Analog decoders typically reach a complexity which prohibits circuit-level simulations of the overall decoding network. Circuit-level simulations are then restricted to small sub-blocks of the overall decoder. Only very small analog decoders may be simulated for a few selected decoding configurations. A more detailed performance evaluation, in particular in terms of the BER, thus needs to rely on high-level simulation models. It is therefore essential to capture the main characteristics of the analog decoder in sufficient enough detail while at the same time reducing the complexity in order to facilitate Monte Carlo simulations for the BER.

In the following we introduce our comprehensive design and simulation environment which has been developed as part of this thesis. This design and simulation environment is illustrated in Fig. 5.1. The basic communication system, including the generation of random code

[1] Note that the pads, bonding, packaging and the printed circuit board (PCB) add more parasitics which may further limit the potential speed of analog decoders.

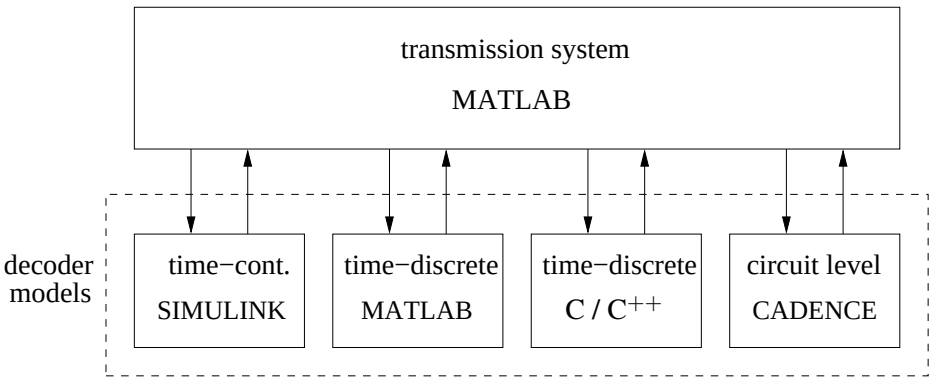



**Figure 5.1**: Design and simulation environment.

words, encoding, BPSK modulation, AWGN channel model and the calculation of the channel values $L_c y$ is implemented in MATLAB. Different decoder models can then be facilitated by MATLAB. These decoder models include time-continuous models in SIMULINK, time-discrete models implemented in C/C$^{++}$ or MATLAB and even circuit-level simulations using the Spectre simulator as part of the CADENCE design environment. Using a configuration file for the Spectre simulator it is also possible to specify a Verilog-A model instead of the transistor-level netlist. This allows us to develop and simulate decoder models on different levels of abstraction within a unified environment. Furthermore, circuit-level parameters can easily be optimized in order to minimize the error introduced by the circuit implementation. Both bottom-up and top-down design approaches are supported.

In order to allow a fair comparison between different simulation models all the following simulations were run on the same set of channel values as the corresponding reference decoder.

### 5.1.1 Time-Continuous Simulation Model

We start with our time-continuous simulation model which allows us to accurately capture the dynamic behavior of analog decoders. Such dynamic networks are conveniently modeled in SIMULINK, an extension toolbox for MATLAB. The time-continuous simulation model consists of a node processor for each directed node in the normal graph followed by lumped resistor-capacitor (RC) elements for each signal component, see Fig. 5.2. This simulation model dates

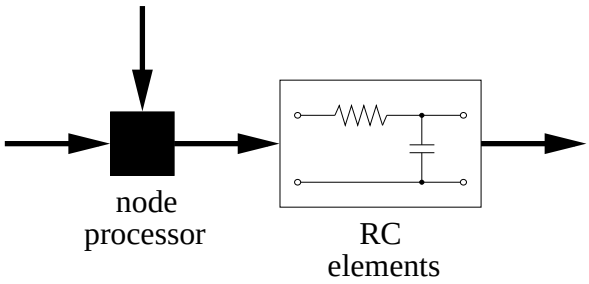



**Figure 5.2**: Time-continuous simulation model for a directed node in the code graph.

back to the early days of analog decoding and was used by Hagenauer *et al.* in, e.g., [Hag98], [Win98], and [HW98]. When each node processor in the decoder performs the exact decoding operations we obtain an ideal analog decoding network. This ideal network is used as our analog reference decoder whenever decoder complexity and thus simulation time permits. The ideal operations in the node processor can easily be replaced by a more enhanced description

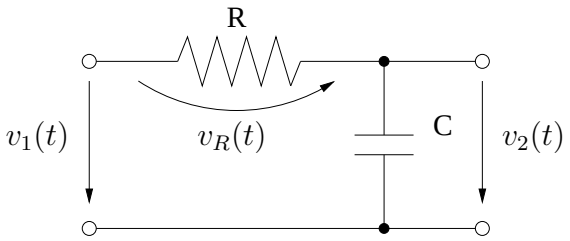



**Figure 5.3**: Basic RC element consisting of a resistor and capacitor.

which captures the characteristics of analog transistor circuits more accurately. Such a model may be derived from and validated against circuit-level simulation results which can easily be obtained within the same simulation environment.

The RC elements include all the parasitics associated with the node processor and the connecting wires to consecutive node processors. We will see later in Chapter 6 that all signals communicated between node processors are voltage signals. The thick lines in Fig. 5.2 refer to vector signals, i.e., represent voltage vectors, where the length of the vectors corresponds to the number of inputs and outputs of the node processor. For each component of this voltage vector there is a dedicated RC element as shown in Fig. 5.3. In the following we analyze such an RC element in more detail. We derive two possible descriptions of RC elements as they are used in the SIMULINK implementation of our simulation model. With $v_1(t) = v_2(t) + i(t)$ R and

$$i(t) = \mathrm{C}\,\frac{\mathrm{d}\,v_2(t)}{\mathrm{d}t} \tag{5.1}$$

we obtain the first order differential equation

$$\frac{\mathrm{d}\,v_2(t)}{\mathrm{d}t} = \frac{1}{\tau}(v_1(t) - v_2(t)), \tag{5.2}$$

with $\tau = \mathrm{RC}$. Given a constant input voltage $v_1(t) = v_1$ we obtain as solution for this differential equation

$$v_2(t) = v_2(t_0)\, e^{-(t-t_0)/\tau} + v_1\left(1 - e^{-(t-t_0)/\tau}\right). \tag{5.3}$$

When we further assume that $v_2(t_0) = 0$ this expression simplifies to

$$v_2(t) = v_1\left(1 - e^{-(t-t_0)/\tau}\right). \tag{5.4}$$

The Fourier transform of (5.2) yields

$$U_1(\omega) = U_2(\omega) + j\omega\tau U_2(\omega), \tag{5.5}$$

with $\omega = 2\pi f$. With the definition of the transfer function

$$H(\omega) = \frac{U_2(\omega)}{U_1(\omega)} \tag{5.6}$$

then follows

$$Re\left\{H(\omega)\right\} = \frac{1}{1 + (\omega\tau)^2}. \tag{5.7}$$

A set of identical RC elements is conveniently described using the state space representation

$$\frac{\mathrm{d}\,\boldsymbol{s}(t)}{\mathrm{d}t} = \boldsymbol{A}\boldsymbol{s}(t) + \boldsymbol{B}\boldsymbol{v}_1(t), \tag{5.8}$$

$$\boldsymbol{v}_2(t) = \boldsymbol{C}\boldsymbol{s}(t) + \boldsymbol{D}\boldsymbol{v}_1(t), \tag{5.9}$$

where $\boldsymbol{v}_1(t)$ and $\boldsymbol{v}_2(t)$ describe the voltage vectors at the input and the output of the RC element, respectively. Equation (5.8) and (5.9) represent the time-continuous equivalent of the state space representation introduced in Chapter 2. With $\tau = \text{RC}$ we obtain for the state matrix

$$\boldsymbol{A} = -\frac{2}{\tau}\,\boldsymbol{I}_N \tag{5.10}$$

and the input matrix

$$\boldsymbol{B} = \frac{1}{\tau}\,\boldsymbol{I}_N, \tag{5.11}$$

where $\boldsymbol{I}_N$ denotes the $N \times N$ identity matrix with $N$ as the the size of the input/output voltage vectors. Furthermore, we obtain $\boldsymbol{C} = 2\boldsymbol{I}_N$ and $\boldsymbol{D} = \boldsymbol{0}$.

For simplicity, the parasitics of all nodes in the code graph are assumed to be identical. Different RC values for the interconnects between analog component decoders are investigated in [Sch05] and Gaussian distributions of the RC delays are considered in [HB04]. However, the results indicate that there is only a negligible impact on decoder performance in terms of BER when the outputs have settled.

A special time-continuous simulation model for tailbiting convolutional decoders is covered in Section 5.2.3.

### 5.1.2 Time-Discrete Simulation Model

The time-continuous simulation model from above leads to a computational complexity which only allows the simulation of small analog decoding networks. For the simulation of larger analog decoders we need to rely on faster simulation models operating in discrete time. More precisely, we do not change the simulation model as such. The difference is how the differential equations defined by the node processor and the RC elements in Fig. 5.2 are solved numerically. The time-continuous SIMULINK model in Section 5.1.1 relies on advanced numerical methods with variable step size in order to numerically solve the differential equation in (5.2). In our time-discrete model we apply the Euler-Cauchy method (see, e.g., [MV91]) with a fixed step size. This one-step algorithm can be expressed as

$$\boldsymbol{v}_2(k+1) = \boldsymbol{v}_2(k) + \frac{\Delta t}{\tau}(\boldsymbol{v}_1(k) - \boldsymbol{v}_2(k)), \tag{5.12}$$

where $k$ is the discrete time index and $h = \Delta t/\tau$ the (variable) step size of the model. For the special case of $\Delta t/\tau = 1$ we obtain

$$\boldsymbol{v}_2(k+1) = \boldsymbol{v}_1(k). \tag{5.13}$$

This means that the output voltage of the RC element in the next time step, i.e., after time $\Delta t = \tau$, equals the input voltage of the RC element. The RC element then degrades to a simple delay element which shifts the output of the node processor by one discrete time instant. The results of the node processor are then available as input to consecutive nodes in the next time step. This special case of the simulation model corresponds to the conventional (time-discrete) message passing algorithm.

Whenever the time-discrete solution in (5.12) yields identical results as the time-continuous decoder model from Section 5.1.1 it can be applied in order to reduce simulation time. The fastest simulations are obtained with $h = 1$, but smaller values of $h$ capture the time-continuous message exchange in the analog decoder more accurately. Further comments on the equivalence of analog and digital decoders can be found in Section 5.5.

### 5.1.3 Circuit-Level Simulations

The most detailed and accurate simulation results of analog decoders are obtained from circuit-level simulations of the overall decoding network. This is in particular true for post layout simulations including the parasitics extracted from the physical layout of the decoder chip. However, such circuit-level simulations are up to several orders of magnitude more complex and time consuming than simulations based on the decoder models in Sections 5.1.1 and 5.1.2. In fact, for most interesting coding schemes these simulations are impractical. In this case only small sub-blocks of the overall decoder can be simulated and verified on circuit-level.

We use the CADENCE design environment which incorporates powerful tools such as schematic capture, layout editing, simulation and physical verification. The circuits are designed using a schematic editor which allows the generation of a netlist file for the Spectre circuit simulator. Besides the netlist file the circuit simulator also requires stimuli and analysis files which define the inputs to the circuit and the type of analysis, respectively. Both are generated using interactive graphical interfaces. The main drawback here is that input stimuli need to be entered manually, which can be tedious when there are a large number of input signals. This requires a lot of manual interaction before a new simulation can be started with a different input configuration. Also, the integrated results browser does not allow a direct comparison with high-level simulation results in MATLAB.

Our new environment in Fig. 5.1 allows a significant acceleration of the design and simulation process. A detailed diagram of our link between the MATLAB and the CADENCE environments is shown in Fig. 5.4. The transistor circuits are designed using the CADENCE

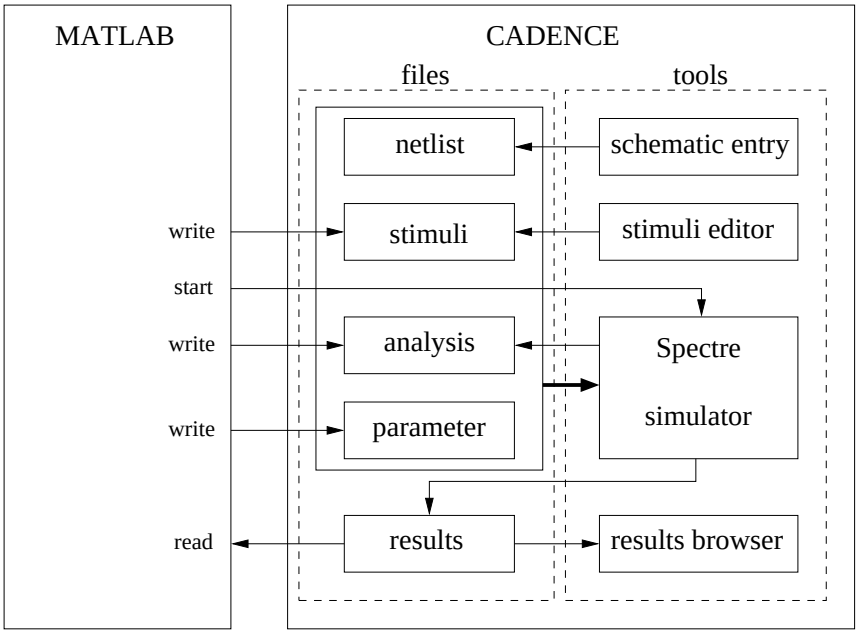



**Figure 5.4**: Link between MATLAB and the CADENCE design environment.

schematic entry tool. The schematic of an analog decoder or a single decoder building block is then translated into a netlist file where important design parameters can be specified as variables. The simulation of this transistor configuration is then fully controlled by MATLAB scripts which generate the input stimuli file, the analysis file and the optional parameter file. The Spectre circuit simulator is then invoked from MATLAB and, after the simulation run is terminated, MATLAB also reads the results file generated by Spectre in order to process and evaluate the simulation results. The circuit simulator thus becomes an additional simulation engine of MATLAB. This new environment allows us to run repeated simulations with different

input signals and to sweep certain design parameters. Different circuit-level parameters can be optimized and the simulation results can directly be validated against the results obtained from high-level simulation models. The direct link between the circuit-level simulator and the system level simulator allows us to develop high-level simulation models based on circuit-level simulation results. We can then evaluate the performance of analog decoders with very accurate simulation models on system level. Possible measures for the the performance are the BER or the trajectories in EXIT charts. The results of these system level simulations can be used in order to optimize certain parameters in the circuit design. Parameter sweeps can then be performed on system level rather than on circuit-level as it is common practice in standard electronic design automation (EDA) tools. This approach appears to be novel. It will be heavily utilized in Section 8.2 in order to optimize the circuit-level parameters of an analog LDPC decoder implemented in CMOS technology.

## 5.2 Basic Analog Decoding Networks

Analog decoding networks are best described by normal graphs. Every node in the normal graph represents a node processor and the interconnects are solely used for communication between node processors. Motivated by our circuit implementation of these node processors, which is presented in the next chapter, we assume that all nodes in the normal graph have degree three. Any node with a degree larger than three can easily be transformed into nodes of degree three as described in Section 3.4. Due to the bidirectional message exchange in the decoder we then transform normal graphs into directed normal graphs. These directed normal graphs then represent a detailed model of the analog decoder, similar to a block diagram.

In the following we describe some basic analog decoding networks for various different codes. We start with check node and variable node decoders which can be used for decoding SPC codes and repetition codes, respectively. These two simple decoders represent the component decoders for all codes with binary code graphs. We then turn our focus to the decoding of simple block codes like the (7,4,3) Hamming code and the (8,4,4) extended Hamming code. We present different decoding networks based on binary and non-binary normal graphs which may include short cycles. In general, different code graphs give rise to different decoder performance. We introduce further modifications of normal graphs which allow us to closely approach the performance of the APP decoder despite small cycles in the decoding network. Finally, we investigate the important case of analog decoders for tailbiting convolutional codes where the code graph includes a single loop of rather large size.

### 5.2.1 Check Node and Variable Node Decoders

The check node and the variable node decoders are the most basic analog decoding networks. They can be employed for decoding of very simple codes like SPC codes and repetition codes, or, can be utilized as component decoders for any code with a binary code graph. These decoders are solely based on the Boxplus operation and the summation of L-values as introduced in Section 4.3.2.

We start with a general check node decoder which can be used for decoding of a $(N, N-1, 2)$ SPC code. This code can be represented by a normal graph with $N-2$ parity-check nodes of degree three as shown in the lower part of Fig. 5.5. Note that this normal graph represents the memory one trellis of the SPC code where the parity bit terminates the code trellis. Each node in this code graph represents a bidirectional check node processor which calculates the outgoing extrinsic values for each of the three branches based on the incoming messages according to (4.17). The directed view of such a degree three check node processor is depicted in the upper part of Fig. 5.5. Here, every check node in the conventional (bidirectional) normal graph translates into three directed Boxplus elements with two inputs and one output.

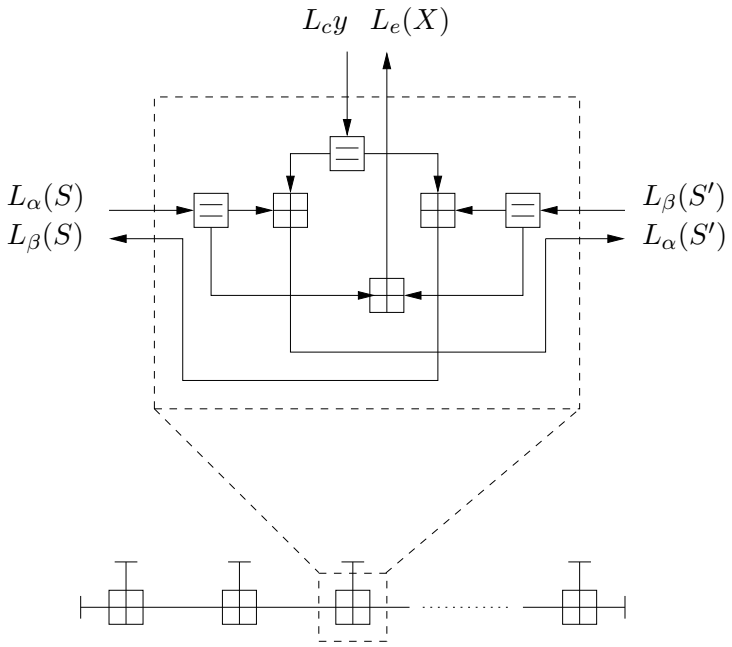



**Figure 5.5**: General check node decoder with a directed view of one check node processor.

The SPC decoder thus requires a total number of $3(N-2)$ such Boxplus operations. The received value $y$ associated with code bit $x$ is multiplied in the receiver with the channel state information $L_c$ in order to obtain the L-value $L_c y$ which is used as decoder input. Further inputs are $L_\alpha(S)$ and $L_\beta(S')$ from the neighboring check node processors. The check node processor then calculates the outputs $L_\alpha(S')$ and $L_\beta(S)$ and passes them on in forward and backward direction along the code graph. Furthermore, the check node processor evaluates the extrinsic decoder output $L_e(X)$ for the corresponding code bit. Additional equality constraint nodes are added in the directed view in order to maintain degree two state variables as required in normal graphs. These equality constraint nodes simply provide two copies of the input signal at the output and do not perform any computations. Such nodes may be used in the analog decoder for the adjustment of certain signal characteristics in order to match the input requirements of the consecutive block. Note that the Boxplus operation of all input values represents a syndrome former which indicates a satisfied parity-check and thus a valid code word as soon as the output becomes positive. This can be realized in the check node decoder with only one additional Boxplus element.

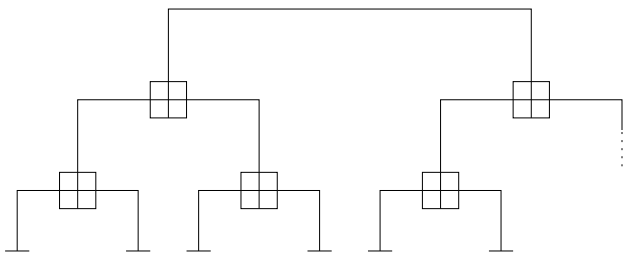


**Figure 5.6**: Check node decoder arranged in a tree structure.

Alternatively, the check node decoder in Fig. 5.5 can also be arranged in a tree structure as shown in Fig. 5.6. The number of blocks and thus the complexity of the analog decoder is identical to the one in Fig. 5.5, but the maximum span of the code graph is reduced. In many cases, such a structure leads to faster analog decoders.

A general variable node decoder is obtained when the parity-check nodes in Fig. 5.5 and Fig. 5.6 are replaced by equality nodes. Such a variable node decoder can be used for decoding a $(N, 1, N)$ repetition code. The $N-2$ nodes in the normal graph then represent bidirectional equality node processors of degree three which calculate the outgoing extrinsic values for each of the three branches based on the incoming messages, see Fig. 4.2. The overall variable node decoder then comprises $3(N-2)$ directed equality node processors with two inputs and one output. Each of these directed equality node processors performs the summation of two L-values so that there are a total number of $3(N-2)$ pairwise summations. Instead of the extrinsic information $L_e(X)$ the variable node decoder can also provide the overall decoder output $L(\hat{X}) = L_c y + L_e(X)$. This can be achieved without any additional complexity by simply changing the wiring in the node processor. Again, the additional equality constraint nodes with one input and two outputs are present in the directed view in order to maintain degree two state variables.

An alternative method for the calculation of the extrinsic output of an equality node processor is to first sum up all incoming messages and then subtract the incoming message from the branch on which the extrinsic information is being sent out, see (4.19). Here, the sum of all incoming L-values requires $N-1$ pairwise summations. The $N$ required subtractions can be treated as summations with a sign inversion of the corresponding inputs. We then obtain a total number of $2N-1$ summations in the decoder for the repetition code. A comparison with the trellis based approach in Fig. 5.5 and the tree based approach in Fig. 5.6 yields that this alternative method is less complex for $N > 5$. An example of such a variable node decoder is given in Fig. 5.7 for the case of a (4,1,4) repetition code with $N = 4$.[2] Fig. 5.7 a) shows the normal graph of the code and Fig. 5.7 b) depicts the directed view of the corresponding decoding network as described above. Note that there are again two different types of equality nodes

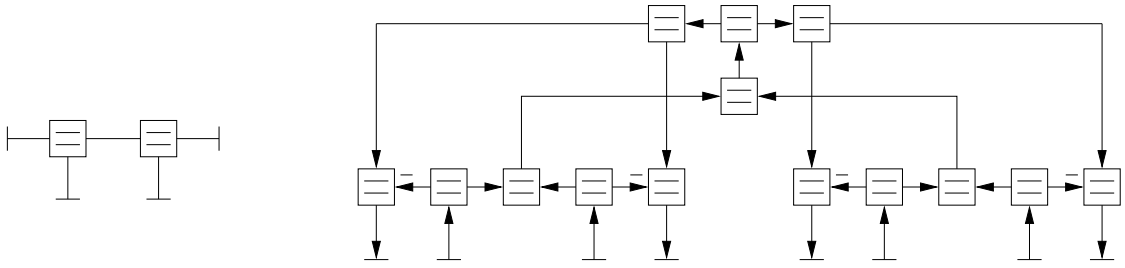



**Figure 5.7**: Variable node decoder in a) with alternative implementation in b).

in Fig. 5.7 b). Equality nodes with two inputs and one output calculate the sum of the corresponding L-values while nodes with one input and two outputs perform no computations. These nodes are used in order to maintain degree two state variables in the normal graph as in Fig. 5.5 and simply provide two copies of the input signal at the output. As already pointed out earlier, such nodes are used in the analog decoder for the adjustment of certain signal characteristics in order to match the input requirements of a consecutive block. The variable node decoder in Fig. 5.7 thus requires a total number of $2N-1=7$ pairwise summations. A minus sign next to the input of an equality node indicates a sign inversion of the corresponding input which can simply be achieved in the circuit implementation by interchanging the wires of the differential input signal. Note that the blocks for the summation of all incoming values are arranged in a tree structure.

Unfortunately, such an alternative computation method is not available for the check node decoder due to the lack of an inverse for the Boxplus operation.

[2]This example is chosen for simplicity and not in order to demonstrate a possible complexity reduction.

These check node and variable node decoders suffice in order to construct analog decoding networks for all codes based on binary graphs, including LDPC codes. Some examples are given in the next sections. The circuit implementation of check node and variable node decoders is covered in Section 6.4.1.

### 5.2.2 Decoders for Simple Block Codes

In this section we investigate different analog decoding networks for simple block codes like the (7,4,3) Hamming code and the (8,4,4) extended Hamming code. These codes are particularly interesting since the corresponding graphs include very short loops. Such small loops in the code graph are known to cause a performance degradation in case of message passing decoding. However, we demonstrate in this section that the presence of small cycles does not necessarily degrade decoder performance. We derive different normal graph representations of these codes which represent different analog decoding networks. These analog decoders are then evaluated in terms of the BER performance by using our different time-continuous and time-discrete simulation models introduced in Section 5.1.

The Tanner graph of the (7,4,3) Hamming code in Fig. 3.1 contains three loops of size four and one loop of size six. In order to obtain a suitable representation of the analog decoding network we transform the code graph into a normal graph with degree three nodes. This is achieved in two steps. We start with the $3 \times 7$ parity-check matrix of the code in (2.20). First, we restrict the degree of the check nodes (rows of $\boldsymbol{H}$) to three by introducing (hidden) state variables. This can be achieved by applying the C3 Algorithm from Section 3.3.1 to (2.20). We then obtain the $6 \times 10$ matrix

| $X_1$ | $X_2$ | $X_3$ | $X_4$ | $X_5$ | $X_6$ | $X_7$ | $S_1$ | $S_2$ | $S_3$ |
|---|---|---|---|---|---|---|---|---|---|
| | 1 | 1 | | | | | 1 | | |
| | | | 1 | 1 | | | 1 | | |
| 1 | | 1 | | | | | | 1 | |
| | | | 1 | | 1 | | | 1 | |
| 1 | 1 | | | | | | | | 1 |
| | | | 1 | | | 1 | | | 1 |

(5.14)

with the three state variables $S_1, S_2$, and $S_3$. Instead of the conventional matrix notation we again use a table format in order to clearly point out the correspondence between the symbol and state variables and the columns of the matrix. Note that only the "1"s in the matrix are shown. In a second step, we restrict the degree of the equality nodes in the normal graph to three by applying the V3 Algorithm from Section 3.4.2. This step only effects column four in (5.14) and leads to the $7 \times 11$ extended parity-check matrix

| $X_1$ | $X_2$ | $X_3$ | $X_4$ | $X_5$ | $X_6$ | $X_7$ | $S_1$ | $S_2$ | $S_3$ | $S_4$ |
|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 1 | | | | | 1 | | | |
| | | | 1 | 1 | | | 1 | | | |
| 1 | | 1 | | | | | | 1 | | |
| | | | | | 1 | | | 1 | | 1 |
| 1 | 1 | | | | | | | | 1 | |
| | | | | | | 1 | | | 1 | 1 |
| | | | 1 | | | | | | | 1 |

(5.15)

using the additional state variable $S_4$. Note that the V3 Algorithm leads to a matrix structure where the column weights for symbol and state variables are limited to two and three, respectively. This implies that the nodes in the corresponding normal graph have a maximum degree

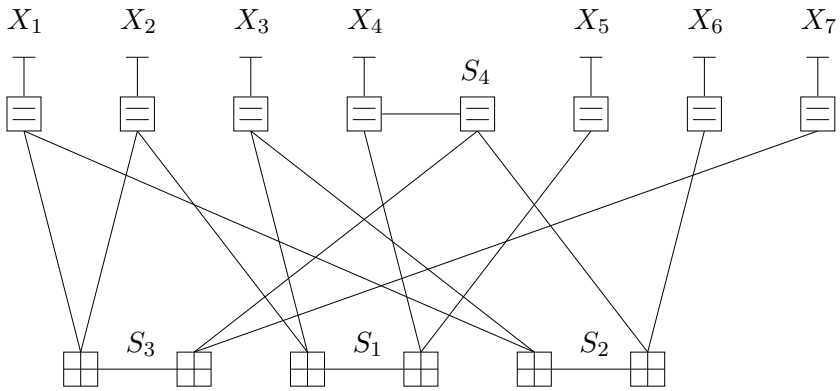



**Figure 5.8**: Decoding network for the (7,4,3) Hamming code based on the $7 \times 11$ extended $\boldsymbol{H}$ matrix.

of three. The corresponding normal graph of the extended $7 \times 11$ parity-check matrix in (5.15) is depicted in Fig. 5.8. Note that the minimum girth has increased from four to six.

We now investigate the BER performance of the analog decoding network for the (7,4,3) Hamming code in Fig. 5.8. The simulation results obtained from our time-discrete simulation model with $h = 1$, i.e., conventional iterative decoding, and $h = 0.1$ are depicted in Fig. 5.9 for 100 and 1000 decoder iterations, respectively. The simulation results are compared with the BER obtained from the time-continuous decoder model in SIMULINK. Note that all three simulation models yield almost identical BER results with marginally better results for the time-discrete simulation model with $h = 0.1$ and the time-continuous simulation model in SIMULINK. All results lie within 0.25 dB of the corresponding APP decoder.

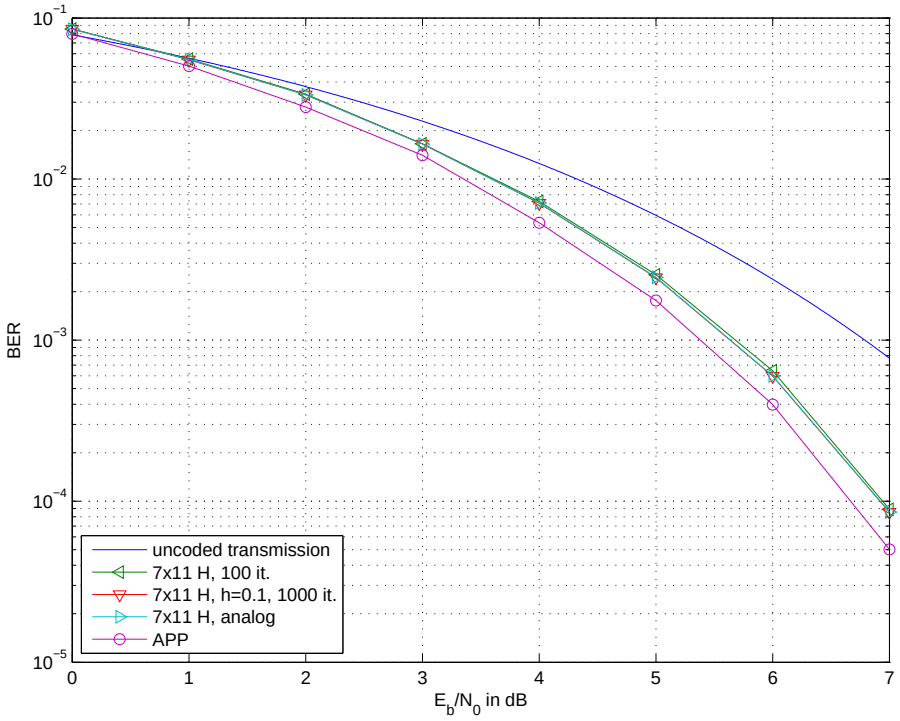



**Figure 5.9**: Decoding of the (7,4,3) Hamming code based on the $7 \times 11$ extended $\boldsymbol{H}$ matrix.

In the following we further modify the normal graph in Fig. 5.8 in a way that decoder performance improves. For this, we introduce a redundant parity-check equation into the parity-check matrix of the (7,4,3) Hamming code in (2.20) which is simply the sum of the three original rows, i.e.

$$\begin{pmatrix} 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix}. \tag{5.16}$$

Instead of applying the C3 and the V3 Algorithm to (5.16) we extend the matrix in (5.15) with the redundant check, i.e., the forth row in (5.16), and then apply the C3 and the V3 Algorithm to this matrix. This leads to the $10 \times 13$ parity-check matrix

| $X_1$ | $X_2$ | $X_3$ | $X_4$ | $X_5$ | $X_6$ | $X_7$ | $S_1$ | $S_2$ | $S_3$ | $S_4$ | $S_5$ | $S_6$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 1 | | | | | 1 | | | | | |
| | | | 1 | 1 | | | 1 | | | | | |
| 1 | | 1 | | | | | | 1 | | | | |
| | | | | | 1 | | | 1 | | 1 | | |
| 1 | 1 | | | | | | | | 1 | | | |
| | | | | | | 1 | | | 1 | 1 | | |
| | | | | | | | | | | 1 | | 1 |
| | | | | 1 | | | | | | | 1 | 1 |
| | | | | | 1 | 1 | | | | | 1 | |
| | | | 1 | | | | | | | | | 1 |

(5.17)

with the two additional state variables $S_5$ and $S_6$. Applying the C3 and the V3 Algorithm to (5.16) would yield a similar matrix which only differs by some row and column permutations. The corresponding normal graph of the parity-check matrix in (5.17) is depicted in Fig. 5.10.

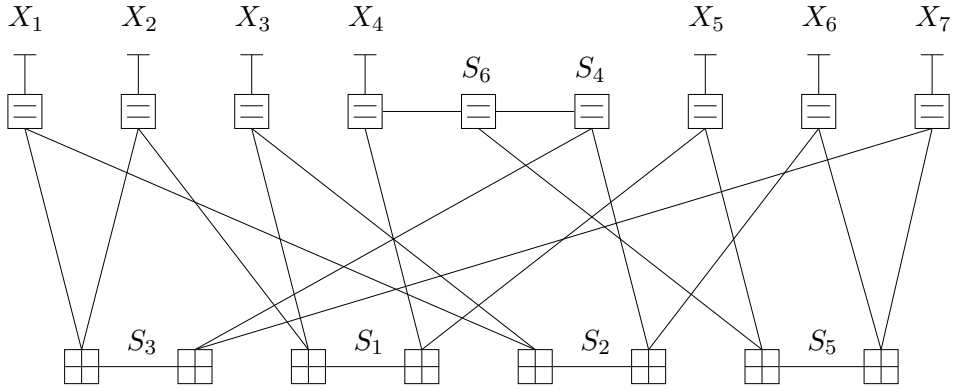



**Figure 5.10**: Decoding network for the (7,4,3) Hamming code based on the $10 \times 13$ extended $\boldsymbol{H}$ matrix with a redundant parity-check.

The BER simulations results for the decoding network in Fig. 5.10 are shown in Fig. 5.11. We notice a significant performance improvement compared to Fig. 5.9 and also compared to the performance of a conventional iterative decoder based on the original $3 \times 7$ parity-check matrix. All three simulation models, i.e., the time-discrete simulation model with $h = 1$ and 100 iterations, the one with $h = 0.1$ and 1000 iterations, and the time-continuous SIMULINK model yield almost identical simulation results. All lie within less than 0.1 dB of the corresponding APP decoder. Again, the iterative decoder with $h = 0.1$ and the time-continuous SIMULINK decoder are marginally better than the conventional iterative decoder based on the same matrix.

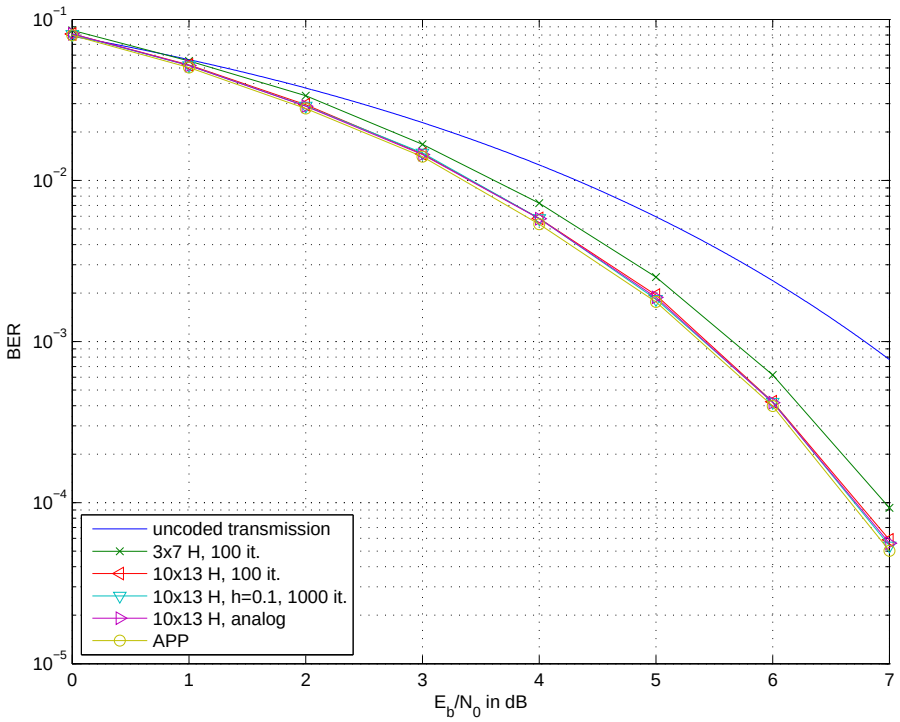



**Figure 5.11**: Decoding of the (7,4,3) Hamming code based on the $10 \times 13$ extended $\boldsymbol{H}$ matrix with a redundant parity-check.

This remarkably good decoder performance is somehow surprising since the normal graph in Fig. 5.10 involves more loops than the one in Fig. 5.8.

A similar good performance of an analog decoder can be achieved with a modified message passing algorithm based on the Tanner graph and a special initialization of some messages [HOMM99].

Analog decoding networks based on graphs with loops are now further investigated for the example of the (8,4,4) extended Hamming code. Similar to the above, we use different code graphs which represent different decoding networks. At the beginning, we restrict ourselves to binary code graphs so that the check node and variable node decoders from Section 5.2.1 can be used as component decoders. The decoding networks are again evaluated in terms of the BER performance by using our different time-continuous and time-discrete simulation models.

We start with an analog decoder based on the $7 \times 11$ extended parity-check matrix as given in (3.19). This matrix leads to the normal graph representation with a minimum girth of six as shown in Fig. 3.8. The simulation results obtained from the time-discrete simulation model with $h = 1$, i.e., conventional iterative decoding, and $h = 0.1$ are depicted in Fig. 5.12 for 100 and 1000 decoder iterations, respectively. These simulation results are compared with the BER performance obtained from our time-continuous simulation model in SIMULINK. All of them lead to virtually identical BER results around 0.2 dB away from the results of the APP decoder. We also plotted the simulation results for a conventional iterative decoder based on the $4 \times 8$ parity-check matrix in (2.22) for comparison. Here, the corresponding code graph has a minimum girth of four. This decoder shows a very poor performance and is around 0.65 dB away from the results for the APP decoder.

Fig. 5.12 also shows the simulation results for the tailbiting wrap-around decoder from Section 4.4.3 based on the tailbiting trellis representation in Fig. 3.5. This tailbiting trellis was derived in Section 3.3 from the MSGM of the (8,4,4) extended Hamming code by using non-

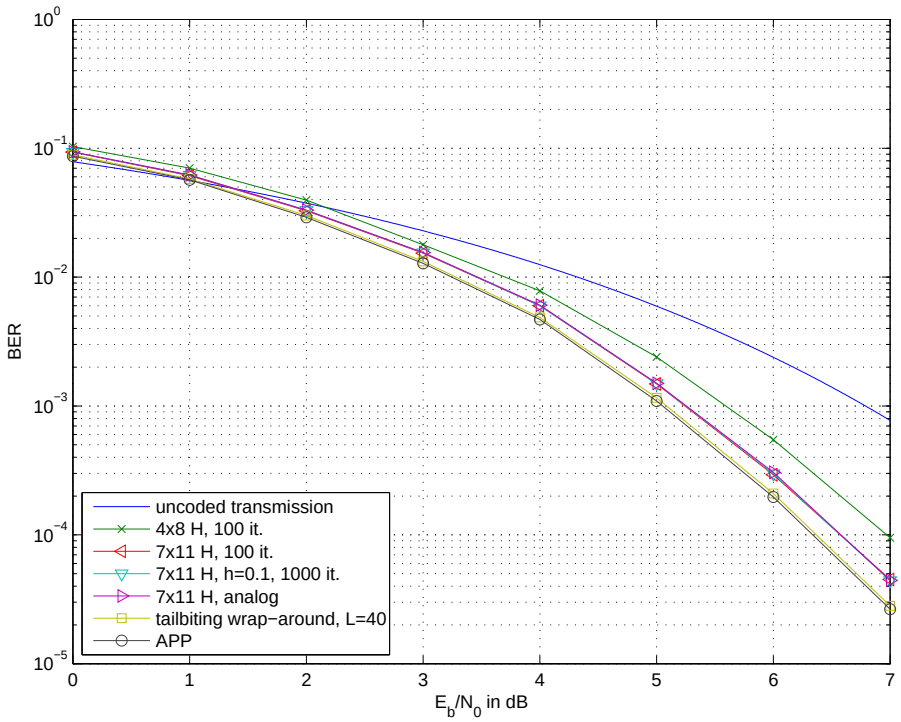



**Figure 5.12**: Decoding of the (8,4,4) extended Hamming code based on different code graphs.

binary state-variables. A stabilization length of $L = 40$ is assumed which corresponds to ten wraps around the tailbiting trellis. It is interesting to note that this decoder closely approaches the performance of the APP reference decoder despite the small size of the tailbiting ring with only four trellis sections.

Note that the use of non-binary state variables also allows the construction of cycle free code graphs, i.e., conventional trellis representations of codes. The decoder based on such a code graph is then a APP decoder. However, code graphs with loops are more interesting to analyze and all code graphs of state-of-the-art turbo codes and LDPC codes involve loops.

### 5.2.3 Tailbiting Convolutional Decoders

Tailbiting convolutional codes play an important role in analog decoding. In fact, most of the analog decoder implementations so far are for tailbiting convolutional codes or codes based on the tailbiting trellis representation, see, e.g., [LHL+99a], [MGYH00], [WDL+01], [ALS+05]. The main advantages of tailbiting convolutional codes are that there is no rate loss due to the termination of the code and that there is no weaker error protection of the last code bits as it is the case when the convolutional code is truncated. This makes them ideal candidates for applications which require codes with short block lengths. The main drawback of tailbiting codes is an increased decoder complexity in the digital domain, see Section 4.4.3. In the analog domain there is no additional complexity in terms of transistor count or area since the required additional computations come for free due to feedback in the fully parallel analog decoding network. The class of tailbiting codes is also particularly interesting because the code graph and thus the analog decoding network involves a single, rather big loop.

The decoding network for a tailbiting convolutional code is depicted in Fig. 5.13. Note that the individual nodes in this code graph correspond to sections in the tailbiting trellis representation of the code. Each node represents an analog node processor which performs a forward and backward recursion as well as the calculation of the decoder output. Two possible implemen-

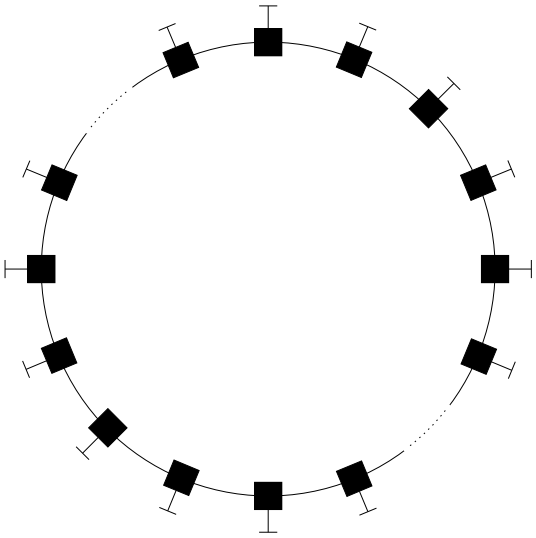


**Figure 5.13**: Decoding network for a tailbiting convolutional code.

tations of such a node processor are depicted in Fig. 5.14. We assume a code rate of $R = 1/2$ so that $L_c y_k^{(1)}$ and $L_c y_k^{(2)}$ are available at the input of the $k$-th node processor. Additional inputs are the $\boldsymbol{\alpha}_k$ and $\boldsymbol{\beta}_{k+1}$ vectors from preceding and succeeding node processors, respectively. The node processor calculates the set of branch metrics $\boldsymbol{\gamma}_k$ which are required for the calculation of the $\boldsymbol{\alpha}_{k+1}$ and $\boldsymbol{\beta}_k$ vectors in the forward and backward recursion of the decoder. Note that these recursions are performed in two separate ring networks which share the same input information from the channel. The decoder output is then calculated based on the results of the $\alpha$ and $\beta$ recursions. Fig. 5.14 a) represents a straightforward implementation of the decoding algorithm as described in Section 4.4.2. This node processor provides either the overall decoder output $L(\hat{U}_k)$ or only the extrinsic information $L_e(U_k)$ depending on the calculations in the output

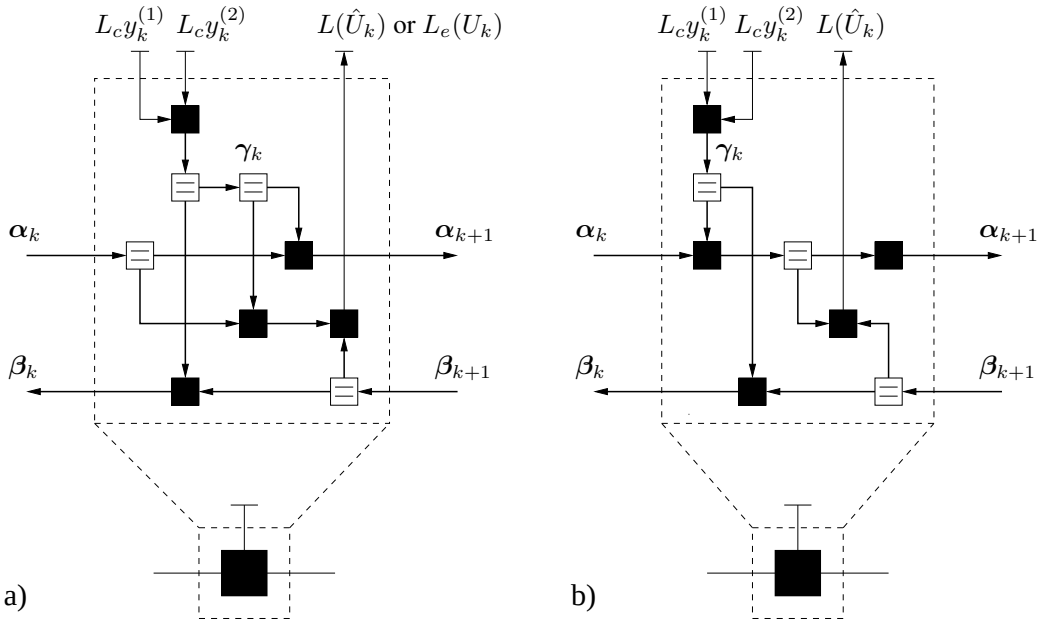



**Figure 5.14**: Implementation of a node processor in a convolutional decoder for codes with systematic feedback encoder in a) and for codes with feedforward encoder in b).

node. The alternative implementation of the node processor in Fig. 5.14 b) can be facilitated for convolutional codes with feedforward encoder where the information bit appears in state $S'$. In this case the output $L(\hat{U}_k)$ can be calculated based on the vectors $\boldsymbol{\alpha}_{k+1}$ and $\boldsymbol{\beta}_{k+1}$ according to (4.37). This typically leads to significantly reduced complexity of the node processor. The circuit implementations of the node processors in Fig. 5.14 are covered in Section 6.4.2.

The ring structure of the tailbiting code allows a very efficient implementation of the time-continuous simulation model in SIMULINK as illustrated in Fig. 5.15. This simulation model

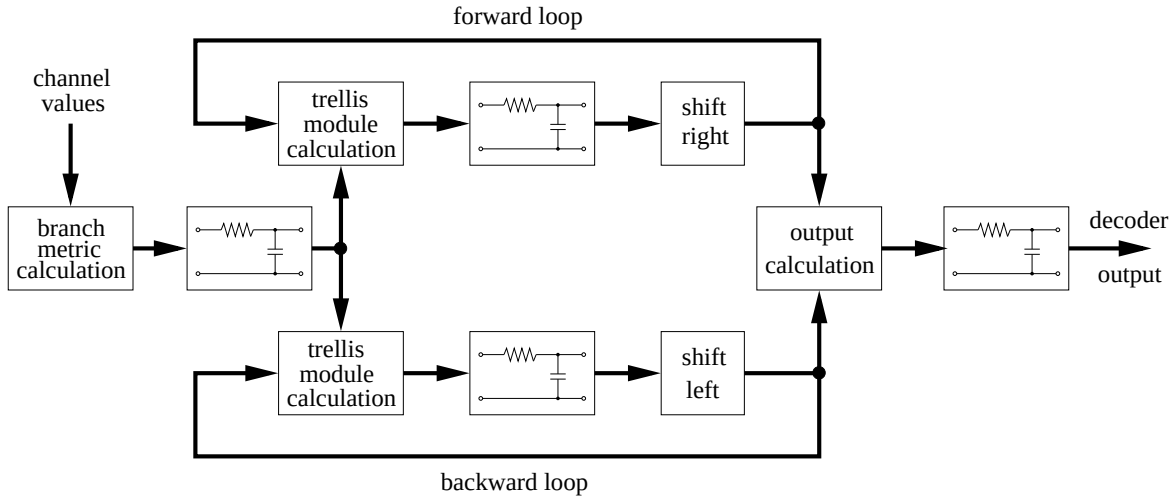



**Figure 5.15**: Time-continuous simulation model of an analog tailbiting convolutional decoder.

requires only one node processor instead of one for each node in the code graph. The individual operations of this node processor involve branch metric computation, the calculations in the two trellis modules, and the calculation of the decoder output. All computational blocks are assumed to perform exact, i.e., ideal, decoder operations. The thick lines in Fig. 5.15 represent large signal vectors which combine the corresponding signal vectors of all node processors. Each module in Fig. 5.15 consequently performs matrix vector operations for the overall block length at the same time. This technique allows the exploitation of fast matrix vector computations in SIMULINK which significantly speeds up the simulations. Note that each computational block in Fig. 5.15 is followed by a lumped RC element which affects all signal components. In order to allow signal propagation along the forward and backward loop in the decoder we need to appropriately shift the outputs of the trellis modules. A shift of the signal vector to the right guarantees that the output of a trellis calculation in the forward recursion is applied to the input of the consecutive trellis section. The shift to the left feeds back the output of the trellis calculation in the backward recursion to the input of the preceding trellis section. The RC elements are conveniently described together with the cyclic shift operation using the state space representation from (5.8) and (5.9). Matrices $\boldsymbol{A}$ and $\boldsymbol{B}$ are taken from (5.10) and (5.11), respectively, and the output matrix $\boldsymbol{C}$ is adopted by applying a circular shift operation. For the forward recursion the matrix $2\boldsymbol{I}_N$ is shifted by one column to the right and for the backward recursion it is shifted by one column to the left. Again, we have $\boldsymbol{D} = \boldsymbol{0}$.

Note that this simulation model requires only one node processor independent of the block length of the code, which allows an easy reuse of this simulation model for codes with different block lengths.

The time-continuous simulation model in Fig. 5.15 can also be utilized for terminated convolutional codes. In this case, the simulation model is slightly modified in a way that the forward and the backward recursions can be initialized appropriately at the beginning and the end of the

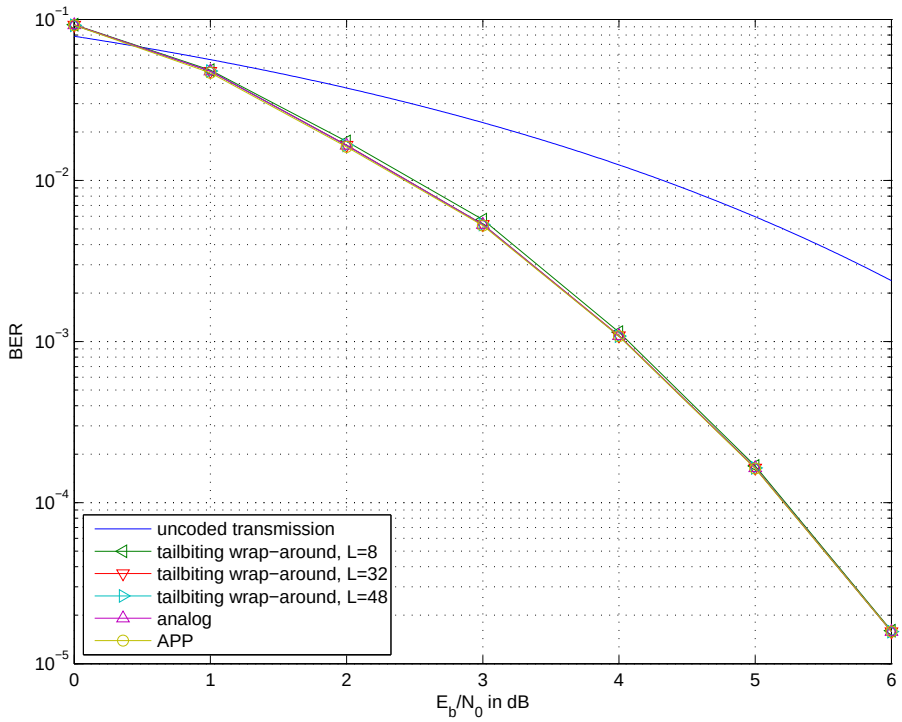



**Figure 5.16**: Decoding of the (32,16) tailbiting code with memory two.

code graph. This effectively cuts the ring structure of the simulation model while the advantage of faster simulations is still maintained. The simulation model then represents an efficient time-continuous implementation of the APP decoding algorithm.

The simulation results for the (32,16) tailbiting convolutional code with memory two and generator polynomials (7,5) are depicted in Fig. 5.16. The time-continuous simulation model in SIMULINK achieves virtually identical BER results as the APP decoder for this code. The results are compared with the performance of the tailbiting wrap-around decoder from Section 4.4.3. A stabilization length $L$ of 8, 32 and 48 is considered, which corresponds to 1/2, 2 and 3 wraps around the tailbiting ring. All three decoder configurations closely approach the performance of the APP decoder. Note that the soft output of the decoder may have a poor quality which is not necessarily reflected in the measured BER. This effect is further investigated later in this chapter by measuring the distributions of the L-values at the decoder output and simulating the characteristic curves of the decoders.

The circuit implementation of the analog decoder for the (32,16) tailbiting convolutional code from above is covered in Section 6.4.2.

## 5.3 Analog Sliding Window Decoding

Various different analog decoder implementations are reported in the literature. Most of these decoders are for codes with a block length of only a few bits and the largest reported analog decoder so far is for a block size of 256 information bits [Win04]. The block size of these decoders is kept small since these decoder chips are intended as proof of concept rather than for a commercial application. However, the feasible block length in analog decoding is also limited by the fully parallel decoder architecture. So far, we assumed that the overall code graph needs to be implemented in analog transistor circuits. This implies that decoder complexity increases roughly linearly with the block length. In most practical applications this approach

leads to an excessive transistor count and consequently a large chip size. Furthermore, a large amount of parallelism in the decoder enables data rates which may not be required by the application. Analog decoding may then not be competitive with digital decoders which are easier to adjust to given requirements of the application. Another shortcoming of fully parallel analog decoder architectures is that a particular decoder chip can only be used for a fixed block length. This practically rules out coding schemes working with different block lengths like in UMTS [ETS00] since every block length would require a dedicated decoder chip.

In this section we present a concept on architecture level which allows us to break the linear relationship between block length and decoder complexity by adopting the sliding window technique well known from digital decoder implementations. Here, parallel decoding of the overall code block is replaced by sequential decoding of typically consecutive code fragments. Decoding then takes place in a comparable small analog decoder core which moves along the code graph until the overall code block is decoded. This approach does not only reduce decoder complexity, but also introduces flexibility into the design process of analog decoders. This is particularly important for the design of area- and power-efficient analog decoders for given speed requirements. Another advantage is that such an analog decoder core can be utilized in different ways so that a variety of different block lengths may be supported. Clearly, the complexity reduction and the increase in flexibility come at the expense of some computational overhead and additional memory may be required in order to store the decoder output.

In the following sections we establish a link between the basic analog sliding window decoder as introduced in [Vei02] and an analog tailbiting convolutional decoder. We analyze the shortcomings of this approach and develop a new solution which improves the soft output and hence the BER performance of analog sliding window decoders. This is achieved in conjunction with a significant reduction in the computational overhead. Some parts of this section have been published in [Moe04a], [Moe06].

### 5.3.1 Basic Concept of the Ring Decoder

The basic concept of analog sliding window decoding relies on a decoder ring as illustrated in Fig. 5.17 a). This decoder implements only a small window of size $W = D + 2L$ trellis sections rather than the overall code trellis. For codes with large block lengths this leads to a significant complexity reduction in the analog decoder. Fig. 5.17 depicts the example of $D = 1$ and $W = 2L+1$. An arbitrary position of the decoding window within the overall code block is shown in Fig. 5.17 b). The beginning and the end of this decoding window are then tied together in order to form a ring structure as shown in Fig. 5.17 c). With a free message exchange at this joint this structure is equivalent to a tailbiting convolutional decoder. In fact, an analog tailbiting convolutional decoder can be used as analog sliding window decoder for a code with the same generator polynomials (either terminated or tailbiting) but a much larger block length. The main difference lies in the way the decoder is loaded and unloaded, i.e., how the decoder inputs are applied and how the decoder output is read. The ring size of the analog tailbiting convolutional decoder in Fig. 5.13 then corresponds to the window size $W$ of the analog sliding window decoder in Fig. 5.17 a). Note that this window size is also related to the window size of the generalized sliding window decoding algorithm in Section 4.4.3.

The operation of the analog sliding window decoder can be described as follows. There are analog storage elements at the input of the sliding window decoder as indicated in Fig. 5.17 a) with small capacitors. We assume that channel values $L_c\boldsymbol{y}_{k-L}, \ldots, L_c\boldsymbol{y}_k, \ldots, L_c\boldsymbol{y}_{k+L-1}$ are already stored and that input $L_c\boldsymbol{y}_{k+L}$ is currently being loaded onto the network. Note that $L_c\boldsymbol{y}_{k+L}$ represents the vector of channel values associated with the node processor at the end of the decoding window. This new input then impacts the messages propagating in both the forward and backward recursion of the decoder. After the settling time of the decoding network we can then read out the decoder output $L(\hat{U}_k)$ from the center of the decoding window. Pro-

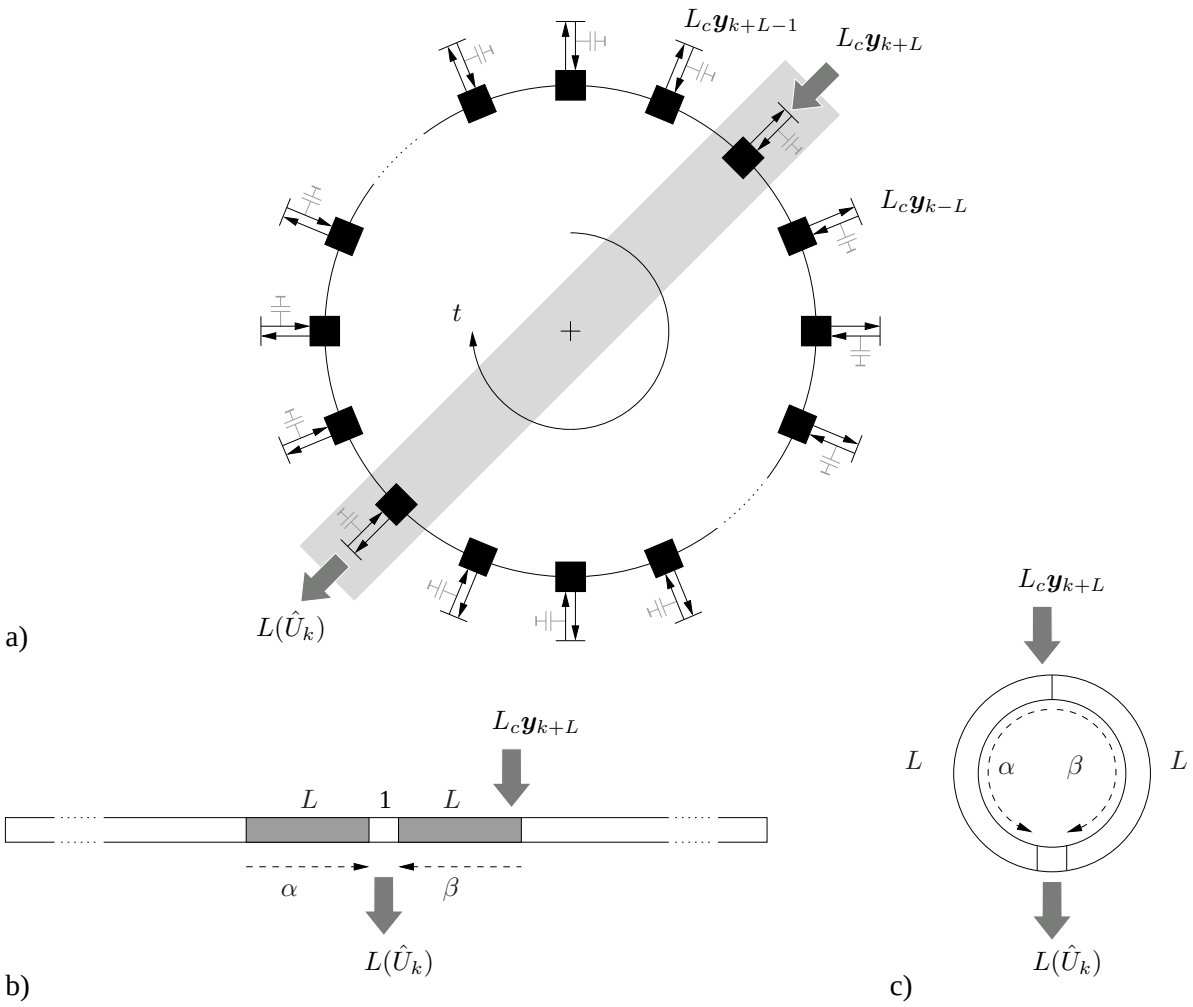



**Figure 5.17**: Basic concept of the ring decoder with $D = 1$ and $W = 2L + 1$.

vided that $L$ is sufficiently large with respect to the memory of the code this leads to a good approximation of the true APP decoder output in both sign and magnitude. This observation is based on the fact that the decoder output is mainly determined by the neighboring $L$ trellis sections to the left and to the right and that any inputs further away do not (significantly) alter the decoder output. Loading and unloading occurs at opposite sides of the decoder ring and both proceed in clock-wise direction around the ring structure as illustrated with the pointer in Fig. 5.17 a). Whenever the position of the decoding window changes another input is loaded onto the network and an old value is overwritten in the cyclic buffer structure. The ring structure of the decoder reduces the number of loading operations to a minimum since the channel values remain stored in the decoding network as long as they contribute to any decoder output. Note that this is in contrast to a digital sliding window decoder where channel values may be reloaded repeatedly.

In general, the parameter $D$, $D \geq 1$, determines the number of trellis sections which are decoded in parallel. It is chosen based on complexity and speed considerations for a particular application. The selection of $L$ directly impacts the performance of the sliding window decoder as we will see later. The ring structure of the decoder implies that the beginning and the end of the decoding window are tied together. This allows the recursions to freely propagate around the ring network without any initialization of the forward and backward recursions at the beginning and end of the decoding window. In this case, the usually used starting distribution, e.g., the uniform distribution of state probabilities, can not be achieved. This requires the sta-

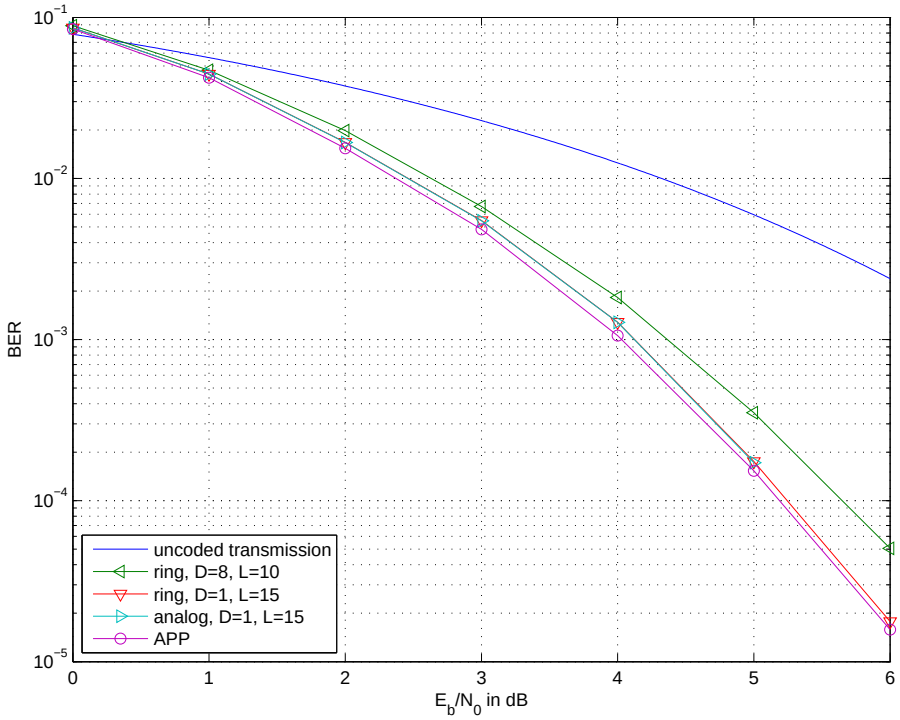



**Figure 5.18**: Decoding of the (512,256) tailbiting convolutional code with different configurations of the ring decoder.

bilization length $L$ to be significantly larger than in the case of an appropriate initialization of the recursions. The lack of such an initialization is also the reason why the decoder produces a large number of bit errors at the beginning and end of terminated convolutional codes.

In the following we analyze the performance of the ring decoder for the example of a (512,256) tailbiting convolutional code with memory two and generator polynomials (7,5). Fig. 5.18 shows the BER simulation results for two different decoder configurations with $D = 8$, $L = 10$ and $D = 1$, $L = 15$ and a ring size of $W = 28$ and $W = 31$, respectively. Note that the selection of $D$ has almost no impact on the BER performance. The parameter $D$ determines the amount of parallelism and thus the speed of the decoder. The two ring decoders are now simulated with the tailbiting wrap-around algorithm from Section 4.4.3. We assume three wraps around the ring of the first decoder configuration and two wraps around the ring of the second decoder. The second decoder configuration is also simulated with our time-continuous simulation model in SIMULINK.[3] We find that both the tailbiting wrap-around algorithm and the time-continuous simulation model yield identical BER results which lie within less than 0.1 dB of the corresponding APP decoder. This is in contrast to the decoder configuration with $D = 8$, $L = 10$ which achieves a clearly suboptimal BER performance.

We now investigate the soft output of the ring decoder. First, we compare the distributions of different type of L-values between a ring decoder with parameters $D = 8$, $L = 10$ and the APP decoder. The distributions of absolute L-values are shown in Fig. 5.19. The dashed curve represents the distribution of the L-values at the decoder input, i.e., $|L_c y|$, which is identical for both decoders. A $E_b/N_0$ value of 3 dB is assumed. The distributions of the extrinsic output $|L_e(U)|$ and the overall decoder output $|L(\hat{U})|$ are illustrated using dash-dotted and solid lines, respectively. We recognize slightly different distributions at the output of the ring decoder than

[3]The simulated BER at the $E_b/N_0$ value of 6 dB has been omitted because of a too long simulation time.

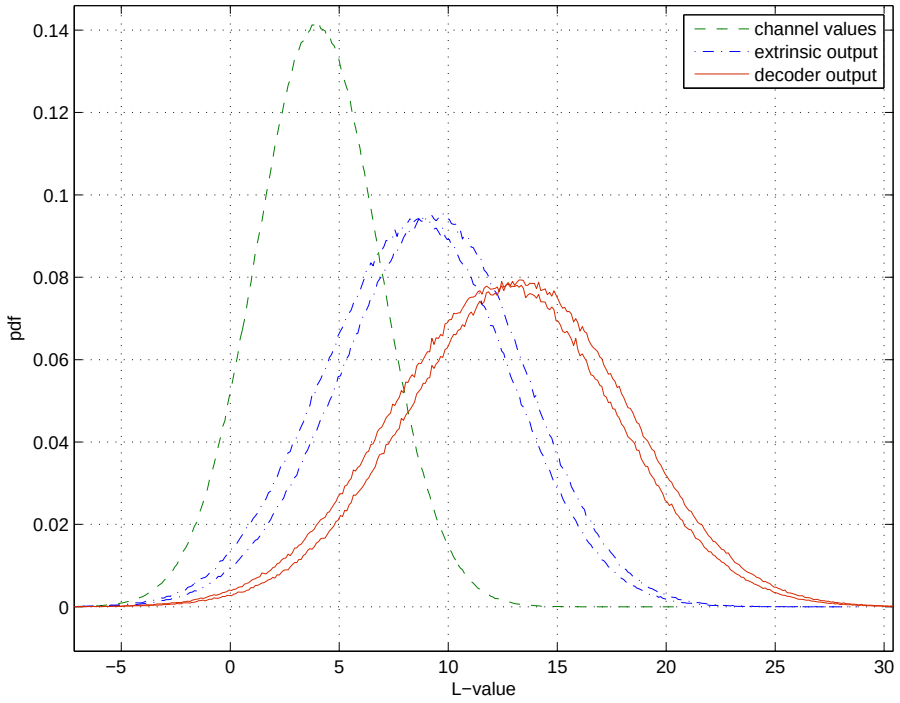



**Figure 5.19**: Comparison between the distributions of absolute L-values of the ring decoder with $D = 8$, $L = 10$ and the APP decoder at a $E_b/N_0$ value of 3 dB.

**Table 5.1**: Mean and variance of the different distributions in Fig. 5.19.

| | APP | | ring decoder $D = 8$, $L = 10$ | |
|---|---|---|---|---|
| type | mean | variance | mean | variance |
| $\|L_c y\|$ | 3.99 | 7.98 | 3.99 | 7.98 |
| $\|L_e\|$ | 9.18 | 17.58 | 8.46 | 17.76 |
| $\|L(\hat{U})\|$ | 13.17 | 25.57 | 12.45 | 25.75 |

for the APP decoder. Mean and variance of the different distributions are summarized in Table 5.1. Note that the ring decoder produces a smaller mean in conjunction with a marginally increased variance for both $|L_e(U)|$ and $|L(\hat{U})|$ which consequently leads to an increased BER. The L-values obtained from the channel clearly satisfy the consistency condition for Gaussian distributions, see (4.49), but this is not the case for the distributions at the decoder output. These distributions are, however, well approximated by Gaussian distributions with the calculated mean and variance from Table 5.1 as it can be seen in Fig. 5.20. Note that the Gaussian assumption at the decoder output does not hold for small SNR values.

The soft output of the ring decoder is now further investigated by means of EXIT chart analysis. Fig. 5.21 shows the characteristic curves of different decoders for the (512,256) tailbiting convolutional code. A $E_b/N_0$ value of 1 dB is assumed. The APP decoder shows the expected behavior and reaches the upper right corner of the EXIT chart. It is surprising to see that the two ring decoder configurations with $D = 1$, $L = 15$ and $D = 8$, $L = 10$ show decreasing $I_E$ values when $I_A$ is increased beyond 0.8 or 0.85. A closer analysis reveals the reason for this bending-off effect of the characteristic curves. The channel values are typically rather small for the considered $E_b/N_0$ values, here with a mean of 2.52. However, large values of $I_A$ produce an a priori input with very reliable L-values, e.g., with a mean of 9.97 for $I_A = 0.95$. Starting

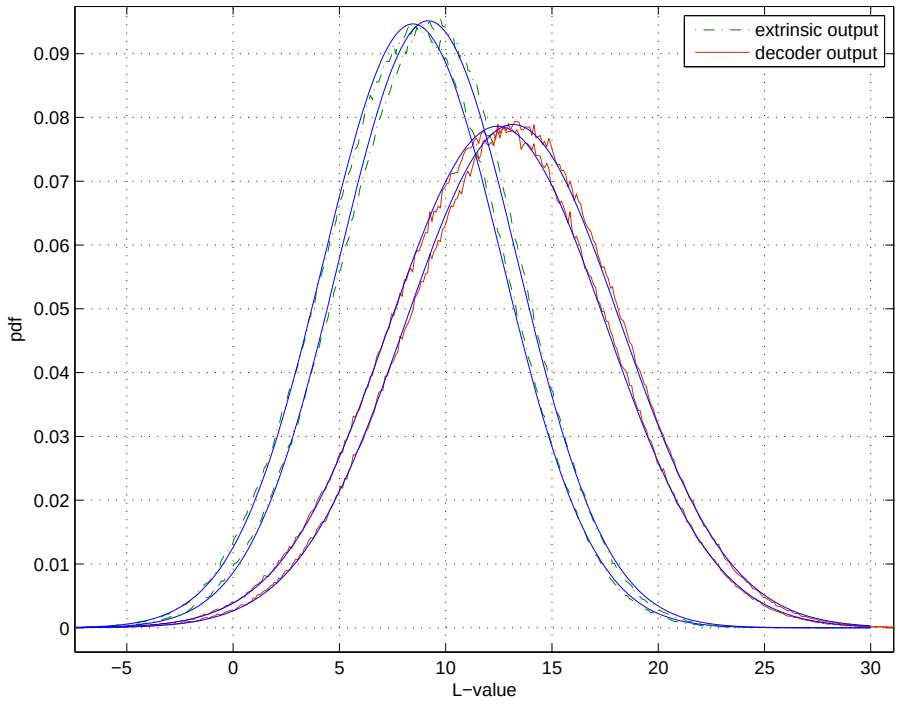



**Figure 5.20**: Comparison between the distributions in Fig. 5.19 and Gaussian distributions according to Table 5.1.

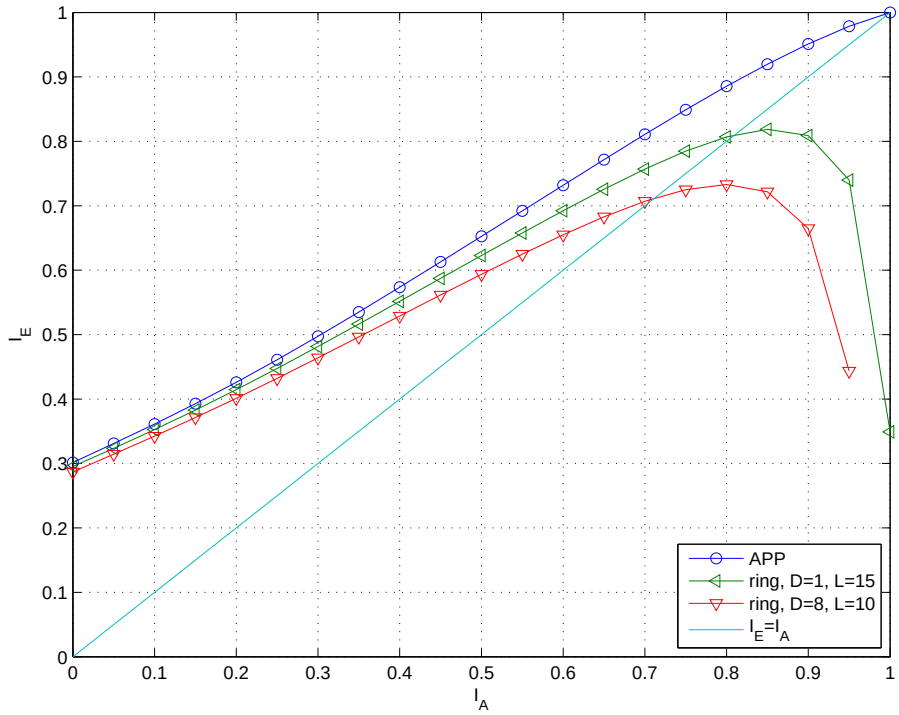



**Figure 5.21**: Characteristic curves of different decoders for the (512,256) tailbiting convolutional code at a $E_b/N_0$ value of 1 dB.

at around $I_A = 0.8$ the output of the ring decoder is sometimes smaller than the L-values at the decoder input. This leads to an extrinsic output $L_e(U) = L(\hat{U}) - L_a(U) - L_c y$ with inverted sign, but only a small magnitude. For $I_A$ close to one this effect occurs more often and may even result in a negative $I_E$. Such a bending-off effect was also found for suboptimal decoders for low-density parity-check convolutional codes [Sch05]. This effect clearly prohibits a good performance in case such a decoder is used as a constituent decoder in a turbo scheme. Note that at least the ring decoder with $D = 1$ and $L = 15$ performs reasonably well in terms of the BER, see Fig. 5.18. This can also be expected due to similar $I_E$ values for $I_A = 0$ in Fig. 5.21.

The soft output of the ring decoder improves with increasing stabilization length $L$. A reasonable good soft output can only achieved when $L$ is increased beyond twice the stabilization length required in case the recursions are initialized appropriately. For small $D$, i.e., a small number of trellis sections decoded in parallel, this leads to a considerable computational overhead which may not be tolerated.

The ring decoder can also be applied to terminated convolutional codes. Here, the performance is rather poor due to a lack of appropriate initializations of the forward recursion and backward recursions at the beginning and the end of the code trellis. The decoder then produces an increased BER at the beginning and the end of the code block which is independent of the stabilization length. In the following section we modify the ring decoder in a way that the recursions can be initialized appropriately within each decoding window. This improves the soft output and also reduces the required stabilization length and thus the computational overhead.

### 5.3.2 Initialization of the Recursions

We now advance the basic concept of the ring decoder in a first step by adding some additional control hardware in order to initialize the forward and the backward recursions in every decoding window. These initializations effectively cut the ring structure of the decoder at the initialization point while the advantages of the cyclic buffer structure are still maintained, i.e., only $D$ trellis segments need to be loaded onto the decoder when the window moves from one position to the next. This type of decoder can be seen as a cyclic implementation of the generalized sliding window decoding algorithm in Section 4.4.3. In the following this decoder is referred to as SwinDec decoder. Fig. 5.22 depicts such a SwinDec decoder with $D = L = W/3$ for the case of a terminated convolutional code. The loading and unloading phases of the first, the second and the last decoding window of the overall code block are shown.

The main difference compared to the basic concept of the sliding window decoder in Section 5.3.1 is that both the forward and backward recursion allow an appropriate initialization at the beginning end the end of the decoding window. The upper part of the figure illustrates the loading phase of the decoder with the channel values, while the lower part shows the output phase of the decoder after the output has settled. In the first decoding window the channel values corresponding to the first $W$ trellis segments are loaded onto the decoder in clockwise direction starting at the top until the cyclic input buffer is filled completely. The forward recursion is now controlled in a way that it starts in the initial encoder state (typically the all-zero state) while the backward recursion is initialized with a uniform distribution. Note that each of the shown rings represents both the forward and the backward recursions of the decoder, which need to be implemented separately. After the settling time of the decoder we can read out the results from the first $D + L$ trellis sections as shown in the lower part of Fig. 5.22 a). Typically, we would read out the results of only $D = W - 2L$ trellis sections, but due to the known starting distribution of the $\alpha$ values at the beginning of the code trellis, no stabilization length is required for the $\alpha$ recursion in the first window. For decoding of the second window in Fig. 5.22 b), the channel values for the next $D$ trellis segments are loaded in clockwise direction onto the decoder in a way that the input buffer corresponding to the first $D$ segments in the first decoding window is overwritten. Now, both the forward and the backward recursion are initialized with

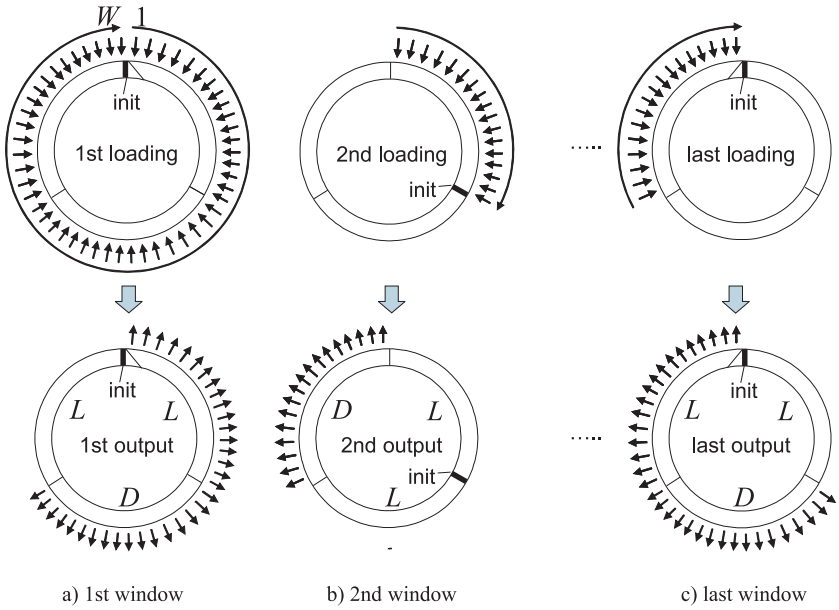



**Figure 5.22**: Loading and unloading of the SwinDec decoder with $D = L = W/3$ for the case of a terminated convolutional code.

a uniform distribution. After the settling time of the decoder we can read out the results for the next $D$ trellis sections from the upper left part of the decoder. Note that the decoder output is always read out $L$ trellis sections behind the decoder input loaded last in order to ensure proper convergence of the backward recursion within the stabilization length $L$. The decoding process continues the same way as demonstrated for the second decoding window for all the other decoding windows until we reach the end of the overall code block. For ease of exposition we assume that the overall block length (including code termination) is an integer multiple of $D$ (and $D = L$). As soon as the channel values for the last $L$ trellis sections are loaded onto the SwinDec decoder, the forward recursion is again initialized with a uniform distribution and the backward recursion is here controlled in a way that it starts in the state of the trellis after termination (usually again the all-zero state). After the settling time of the decoder network we can read out the results for the remaining $D + L$ trellis sections, since no stabilization length is required for the backward recursion at the end of the code block due to the termination of the code. An unwrapped and more detailed view of the decoding process is given in Fig. 5.23, which illustrates the separate recursions within the SwinDec decoder.

The SwinDec decoder can also be deployed for tailbiting convolutional codes. In this case the stabilization length $L$ is required for the forward and the backward recursions of all decoding windows including the first and the last one. There are $D$ trellis sections decoded within each window and the recursions are always initialized with a uniform distribution. The first $L$ trellis sections of the code can only be decoded after the channel values corresponding to the last $L$ trellis sections of the tailbiting code have been received. Similarly, the channel values corresponding to the first $L$ trellis section need to be stored for decoding the last $L$ trellis sections of the tailbiting code. This is because these values are required in order to build up reliable values in the forward and the backward recursion at the beginning and the end of the tailbiting trellis, respectively.

The appropriate initialization of the recursions in the SwinDec decoder dramatically improves the BER performance for terminated convolutional codes. Furthermore, the stabilization length $L$ reduces by more than a factor of two for both terminated and tailbiting convolutional codes, i.e., to 5-6 times the encoder memory, as it is used in digital sliding window decoding. The computational overhead is reduced accordingly. Since the settling time of the SwinDec

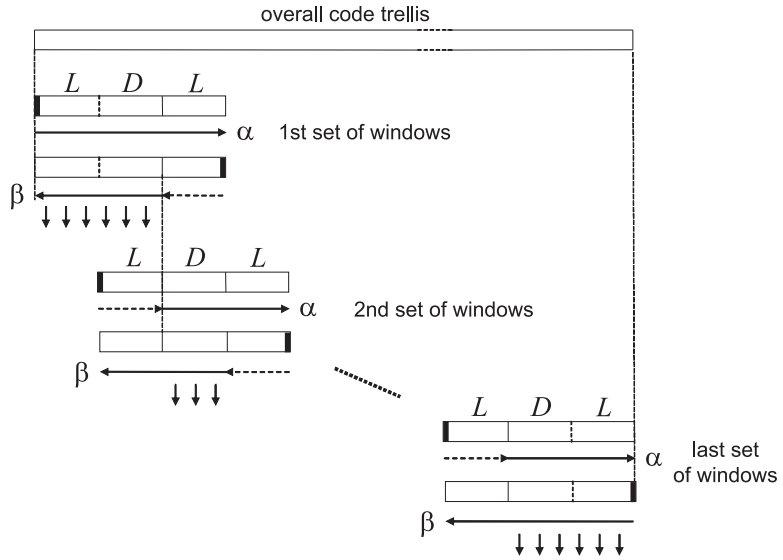



**Figure 5.23**: Unwrapped view of the SwinDec decoder in Fig. 5.22 showing the forward and backward recursions.

decoder directly depends on the stabilization length $L$ this leads to analog decoders which are more than two times faster and less complex. Alternatively, for a given window size $W$, i.e., a given decoder complexity, $D$ can be increased so that more bits are decoded in parallel. This also increases the speed of the decoder.

### 5.3.3 Offset of Forward and Backward Ring

The complexity of the SwinDec decoder as introduced in the previous section can be further reduced by allowing an offset between the forward and backward ring of the decoder. This step solely reduces computational complexity and has no impact on decoder performance. We start with the SwinDec decoder as shown in Fig. 5.22 where the window size amounts to $W = 2L+D$ trellis sections. There are independent forward and the backward recursions within this decoder as illustrated in Fig. 5.23 and each of them needs to be implemented separately. Note that the windows for the forward and backward recursions are lined up so that both recursion are performed for the overall window size. However, we can notice in Fig. 5.23 that it is sufficient to perform these recursions only for $D + L$ trellis sections instead of the overall window size of $W = D + 2L$ sections. This is because the forward and backward recursions require a stabilization length only on the left and right hand side of the decoding window, respectively. The results of the computations for the remaining $L$ trellis sections do not contribute to the decoder output and the calculations are therefore redundant. We can thus remove these $L$ trellis sections in each recursion and work with a window size of only $W' = D + L$ trellis sections.

Fig. 5.24 depicts such a modified SwinDec decoder with $D = L = W'/2$ for the case of a terminated convolutional code. The loading phases for the forward and backward rings are shown separately for the first, the second and the last decoding window of the overall code block. The unwrapped view of this decoder is depicted in Fig. 5.25. We notice that the two rings for the forward and backward recursions in Fig. 5.24 are now shifted against each other. In the first decoding window the channel values corresponding to the first $D$ trellis segments are loaded onto the forward ring in clockwise direction. Similarly, the channel values for the first $W' = D+L$ trellis segments are loaded onto the backward ring in clockwise direction. We assume that both recursions are appropriately initialized as described in the previous section. After the settling time of the decoder we can read out the results from the first $D$ trellis sections as shown in Fig. 5.25. In the second decoding window the channel values for the next $L$ and $D$

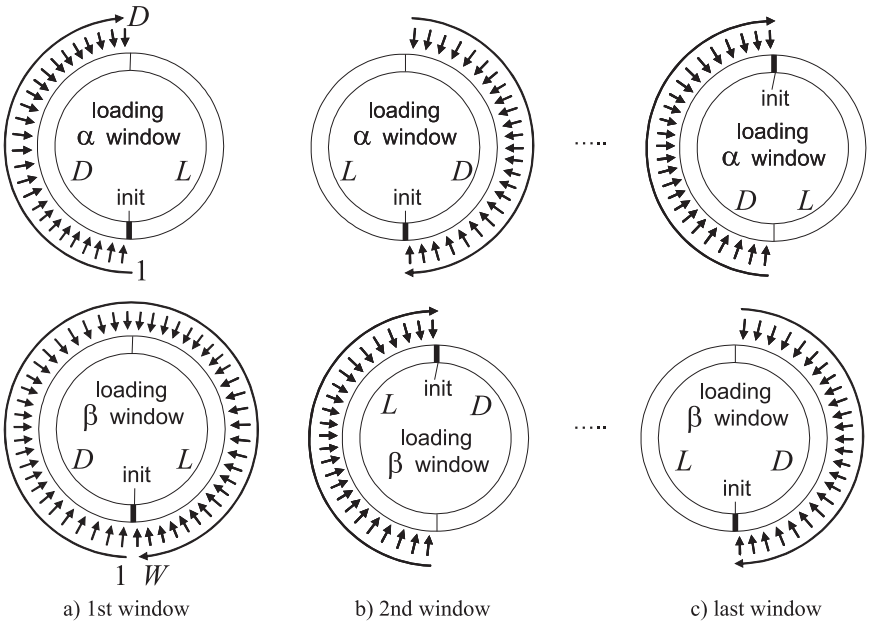



**Figure 5.24**: Loading of the modified SwinDec decoder with $D = L = W'/2$ for the case of a terminated convolutional code.

trellis segments are loaded in clockwise direction onto the forward and backward decoder ring, respectively. The forward ring is now loaded completely while the $D$ segments in the backward ring of the first decoding window are updated with new channel values. Note that for the special case of $D = L$ the role of the corresponding decoder sections are interchanged when we move on from one decoding window to the next. After the settling time of the decoder we can read out the results for the next $D$ trellis sections as shown in Fig. 5.25. The decoder output is always read out $L$ trellis sections behind the decoder input loaded last onto the backward ring in order to ensure proper convergence of the backward recursion within the stabilization length. The decoding process continues the same way as demonstrated for the second decoding window for all the other decoding windows until we reach the end of the overall code block. For ease of exposition we assume that the overall block length (including code termination) is an integer multiple of $D$. After the settling time for the last decoding window we can read out the results

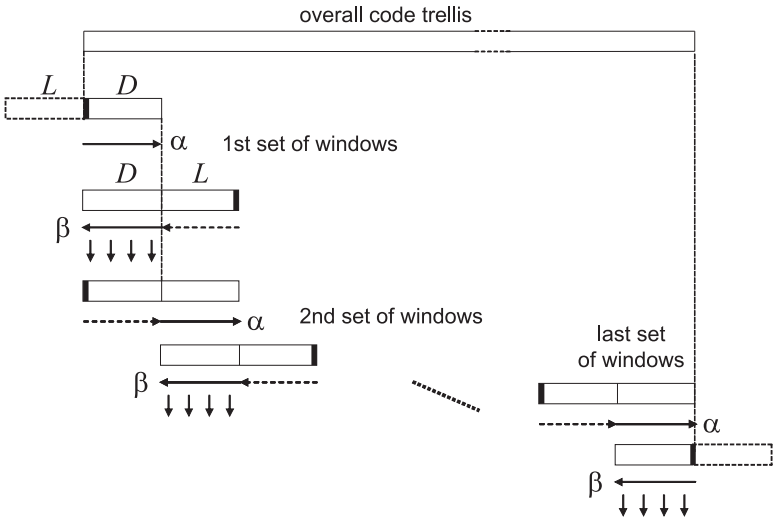



**Figure 5.25**: Unwrapped view of the modified SwinDec decoder in Fig. 5.24 showing the offset between the forward and backward windows.

for the remaining $D$ trellis sections.

Similar to Section 5.3.2, the modified SwinDec decoder can also be deployed for tailbiting convolutional codes. In this case, the forward recursion in the first decoding window and the backward recursion in the last decoding window also require a stabilization length of $L$ trellis sections. All decoding windows are then initialized with a uniform distribution. The first $L$ trellis sections of the code can only be decoded after the channel values corresponding to the last $L$ trellis sections of the tailbiting code have been received. Similarly, the channel values corresponding to the first $L$ trellis section need to be stored for decoding the last $L$ trellis sections of the tailbiting code. This is because these values are required in order to build up reliable values in the forward and the backward recursion at the beginning and the end of the tailbiting trellis, respectively.

This modified version of the SwinDec decoder allows a reduction in the computational complexity of the SwinDec decoder by a factor of $(2L + D)/(L + D)$ compared to the decoder in Section 5.3.2 while maintaining the same decoder performance in terms of BER and speed. In case separate storage elements are used for the forward and backward recursions this approach requires $2(D + L)$ such elements instead of $D + 2L$. However, the reduction in computational complexity is more significant. The complexity of the SwinDec decoder can be reduced further when we allow arbitrary initializations of the forward recursion instead of only the two initializations discussed in Section 5.3.2. We can then use the result of the forward recursion in one decoding window in order to initialize the forward recursion in the next decoding window. This enables the window for the forward recursion to move on continuously without any stabilization length. The forward recursion then requires only $D$ trellis sections instead of $L + D$. Compared to the decoder in Section 5.3.2 the computational complexity is then reduced by a factor of $2(2L + D)/(2D + L)$ and the computational overhead in the SwinDec is identical to digital sliding window decoding.

In the following we analyze the performance of the SwinDec decoder for the (512,256) tailbiting convolutional code with memory two and generator polynomials (7,5). Note that the performance of the SwinDec decoder in Section 5.3.2 is identical to the modified SwinDec decoder presented in this section. Fig. 5.26 shows the simulated BER for a SwinDec decoder with parameters $D = 8$ and $L = 10$. We find that this decoder configuration clearly outperforms the ring decoders in Fig. 5.26 and closely approaches the BER performance of the corresponding APP decoder. The characteristic curves of different SwinDec decoder configurations with $D = 8$ and a stabilization length $L$ of 10, 12, and 15 are depicted in Fig. 5.27. A $E_b/N_0$ value of 1 dB is assumed. It is apparent that the soft output of all configurations is significantly better than for the ring decoders in Fig. 5.21. For $I_A = 0$ all decoders yield almost identical $I_E$ values as the corresponding APP decoder. This observation is consistent with the results in Fig. 5.26 where the configuration with the shortest stabilization length already closely approaches the BER performance of the APP decoder. With increasing $I_A$ values the characteristic curves for the SwinDec decoders are slightly shifted towards lower $I_E$ values.

A more detailed EXIT chart analysis of the SwinDec decoder can be found in Section 5.4.2 where it is used as component decoder for the parallel concatenation of convolutional codes.

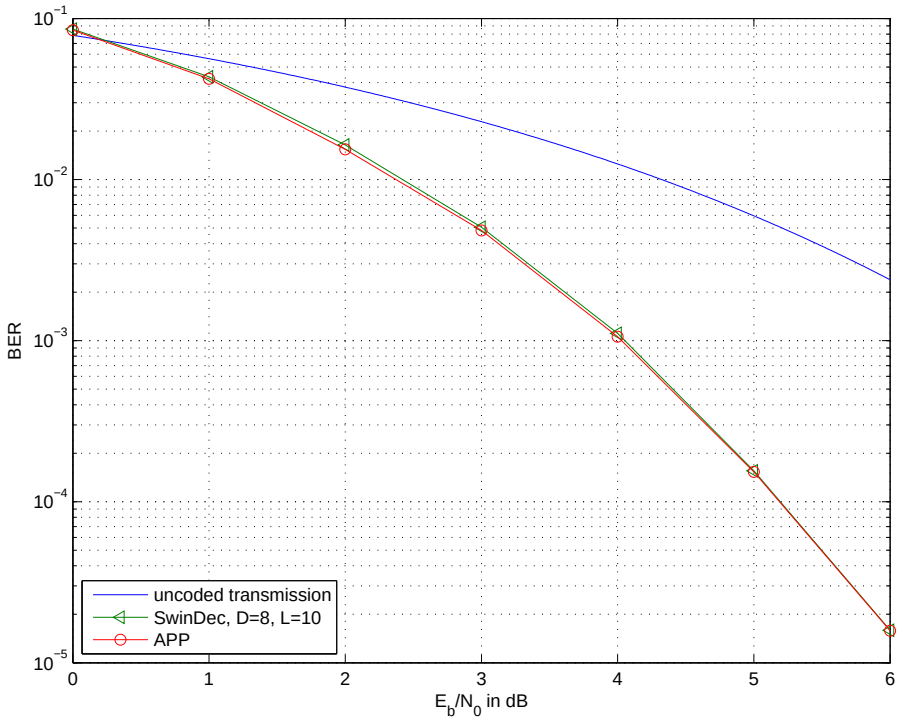



**Figure 5.26**: Decoding of the (512,256) tailbiting convolutional code with the SwinDec decoder.

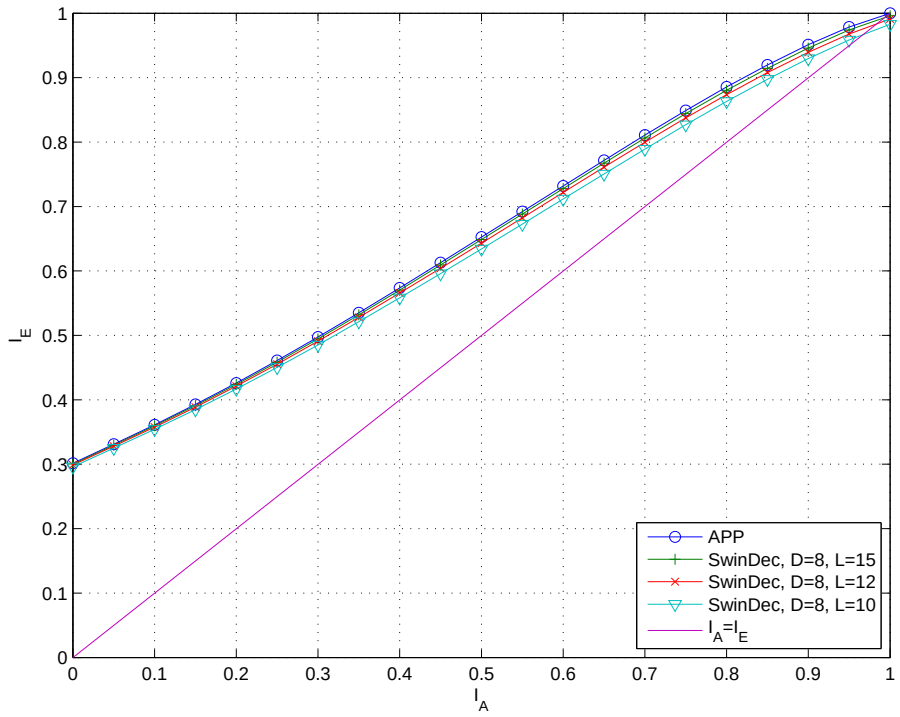



**Figure 5.27**: Characteristic curves of different SwinDec decoder configurations for the (512,256) tailbiting convolutional code at a $E_b/N_0$ value of 1 dB.

## 5.4 Decoder Architectures

After the introduction of some basic analog decoding networks in Section 5.2 and the SwinDec decoder in Section 5.3 we now turn our focus to more powerful analog decoders for turbo codes and LDPC codes. These two classes of codes are known to provide excellent error correcting performance when iterative decoding algorithms are utilized. In the following we distinguish between fully parallel decoder architectures where the overall code graph is implemented in analog transistor circuits and an architecture where only a fragment of the overall code graph is implemented.

Fully parallel decoder architectures are very attractive for turbo codes and LDPC codes. Here, the component codes are decoded concurrently and there is a time-continuous message exchange between the analog component decoders. There are no internal storage elements or clock signals required and the information exchange is fully asynchronous without any iterations. This architecture provides the maximum throughput of the decoder due to the maximum amount of parallelism in the decoding network. It is particularly suited for codes with short block length since decoder complexity increases roughly linearly with the block length of the code. With increasing block length of the code such analog decoders can be up to several orders of magnitude more complex than the decoders described in the previous sections. This poses problems not only for the implementation of analog decoders but also for the simulation. Fully parallel decoder architectures are covered in Section 5.4.1 and Section 5.4.3 for the example of turbo codes and LDPC codes, respectively.

An alternative to such a fully parallel decoder architecture is presented in Section 5.4.2 for the example of a sliding window turbo decoder. In this case, fully parallel processing on the overall code graph is replaced by sequential decoding of sub-blocks of the code. For this the SwinDec decoder from Section 5.3 is employed for decoding the component codes. This allows a significant reduction in the overall decoder complexity at the expense of computational overhead and additional storage elements in the decoder for the extrinsic information. Further advantages of this architecture are that the SwinDec decoder can handle various different block lengths and the speed can easily be scaled with the window size. The use of storage elements allows a flexible realization of different interleaver structures by simply changing the way the storage elements are addressed. In this architecture the message exchange between the analog component decoders is again time-discrete and iterative.

### 5.4.1 Fully Parallel Turbo Decoder

We start with a fully parallel turbo decoder architecture for the parallel concatenation of two convolutional codes. The normal graph of such a turbo code is depicted in the lower part of Fig. 4.9. The nodes in this code graph represent analog node processors and the connectivity of the analog decoding network is defined by the edges of the code graph. The set of the upper and lower analog node processors represent the two component decoders. These component decoders are separated by equality node processors and the interleaver $\Pi$ (or de-interleaver depending on the direction). The bidirectional message exchange within the first component decoder and between the two component decoders is exposed in the upper part of Fig. 4.9 for the example of rate $R = 1/2$ component codes. A more detailed view of a node processor in the component decoders is given in Fig. 5.14 a). The fully parallel analog turbo decoder performs concurrent message passing on the overall code graph according to a flooding schedule without any timing or iterations. This process is fully asynchronous and does not require any internal clock signals. Any node processor in the decoding network immediately passes on its dynamically changing extrinsic output to neighboring nodes, even if they are located in the code graph of the other component code. The neighboring nodes directly benefit from any new information and themselves update their extrinsic output and pass it further on to other nodes. The propagation delay between the blocks and thus the speed of the decoder is only limited by

parasitic resistors and capacitors within the transistors and in the wiring network. The parasitics are modeled in the simulations as lumped RC elements as described in Section 5.1.1. This message exchange in the analog turbo decoder is in contrast to iterative decoding where during the first half iteration the first component decoder calculates the extrinsic output in one full step and then passes it on all at once to the other component decoder. The second component decoder then performs the corresponding decoding operations during the second half of the iteration and passes back its extrinsic information to the first decoder. The fundamentally different message passing in the analog turbo decoder raises the question about the equivalence of the decoder output and the BER performance. This is investigated in more detail shortly.

Fully parallel decoder architectures facilitate decoder implementations providing maximum throughput. Given the maximum amount of parallelism in the decoding network we can reduce the power (and thus speed) per node processing compared to the sequential approach covered in the next section. All extrinsic values are inherently stored in the analog decoding network so that no storage elements are required for the extrinsic information. However, complexity and thus area increase roughly linearly with the block length. This means that this decoder architecture is only suited for blocks lengths of no more than a few hundred bits.

In the following we evaluate the performance of the fully parallel turbo decoder architecture for the example of two turbo codes. Fig. 5.28 shows the simulation results for the (192,64) turbo code originating from the parallel concatenation of two (128,64) tailbiting convolutional codes with generator polynomials (7,5). The performance of the analog turbo decoder is compared with a conventional iterative decoder with 10 and 20 iterations. We find that the BER performance of the iterative decoder approaches the performance of the analog turbo decoder with increasing number of iterations. The second example is for a (192,96) turbo code as it is standardized for the return channel in DVB [ETS03]. It is based on the parallel concatenation of two (144,96) tailbiting convolutional codes with memory three and rate 2/3. The simulation results for the analog turbo decoder and a conventional iterative decoder with 10, 15 and 20 iterations are compared in Fig. 5.29. Again, the BER performance of the iterative decoder approaches the performance of the analog turbo decoder with increasing number of iterations. Note that a reduced number of iterations is analogous to a shorter time allowed for the settling of the outputs in the analog turbo decoder. Both increases the BER.

These presented turbo decoders already reach a complexity which limits the use of our time-continuous simulation model from Section 5.1.1. The simulation time for large SNR values increases here to several weeks on a Pentium IV processor. The BER results for the analog turbo decoder in Fig. 5.28 and Fig. 5.29 are therefore omitted at 3.5 dB. The simulation of higher SNR values or more complex coding themes thus needs to rely on faster time-discrete simulation models.

Similar simulation results as in the above are reported for a slightly modified DVB code in [ALJS04], [ALS+05] and other codes in [HMO00b]. A time-discrete simulation model for turbo codes with $h < 1$ was investigated in [MA00] without finding a considerable gain. A fully parallel turbo decoder is also investigated in [XVG+02] and in the context of concatenated magnetic recording schemes in [AMNX02]. An example of a fully parallel turbo decoder with programmable interleaver can be found in [GGG02], [GG03b], [GG03a].

### 5.4.2 Sliding Window Turbo Decoder

An interesting alternative to fully parallel turbo decoders are sliding window turbo decoders. This type of turbo decoder relies on SwinDec component decoders as introduced in Section 5.3 in order to decode the component codes. There are several advantages of such a decoder architecture. First, decoder complexity can be significantly reduced since only a small fraction of the overall code graph needs to be implemented. This complexity reduction comes at the

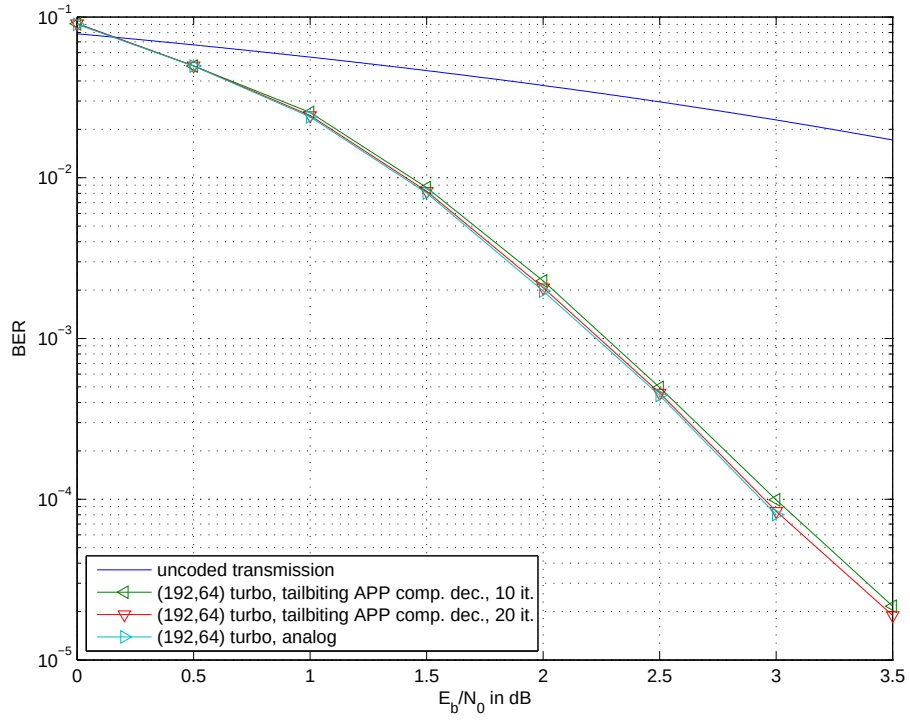



**Figure 5.28**: Decoding of the (192,64) turbo code with memory two tailbiting codes as component codes.

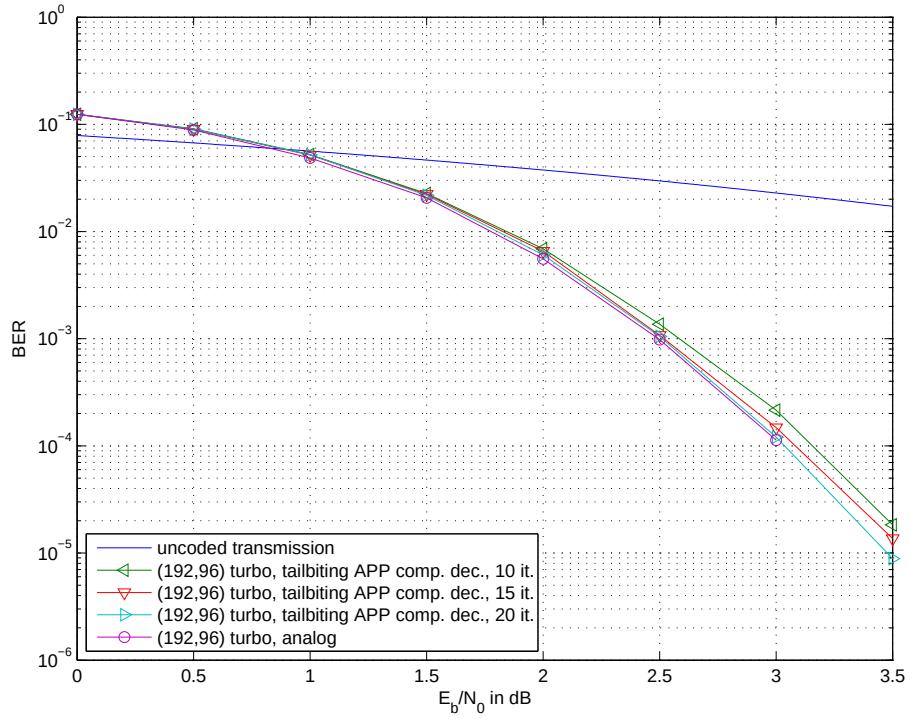



**Figure 5.29**: Decoding of the (192,96) DVB turbo code with memory three tailbiting convolutional codes as component codes.

expense of additional storage elements for the extrinsic information of the component decoders. These storage elements can be implemented either in the analog or the digital domain. In the latter, additional A/D and D/A conversions are required at the extrinsic output and the a priori input of the component decoder. The design of these A/D and D/A converters is extremely important since they are required to operate at a multiple of the block rate. An example of a very area- and power-efficient realization of such a D/A converter is presented later in Chapter 8. The use of storage elements has the advantage of realizing different interleaver structures simply by changing the way the storage elements are addressed. This is not possible with the hard-wired interleaver of a fully parallel decoder. Another advantage is that the SwinDec component decoders can be used very flexibly in order to decode various different block lengths. This feature is mandatory for coding schemes like in UMTS [ETS00] since it is not feasible to use a dedicated turbo decoder implementation for each standardized block length. Also, the speed of the component decoder and thus the speed of the turbo decoder can be scaled with the window size of the SwinDec according to the requirements of the application. Note that the information exchange between the analog component decoders is here iterative which makes this architecture easier to analyze.

In the following we investigate the performance of a sliding window turbo decoder for the example of the (1932,640) UMTS turbo code with rate $R = K/(3K + 12)$ and $K = 640$ information bits. Before we investigate the BER performance of different turbo decoder configurations we start with an EXIT chart analysis of the component decoders as shown in Fig. 5.30 for a $E_b/N_0$ value of 0.5 dB. The characteristic curves of different SwinDec decoders are depicted. All configurations decode a fix number of $D = 8$ information bits within each decoding window and the stabilization length $L$ is chosen to be either 16, 20 or 24. The corresponding curves for an APP decoder are added as reference. We notice that with decreasing values for the stabilization length and increasing $I_A$ values the SwinDec decoder output generates lower $I_E$ values as the APP decoder. This narrows the available tunnel for the trajectory and thus leads

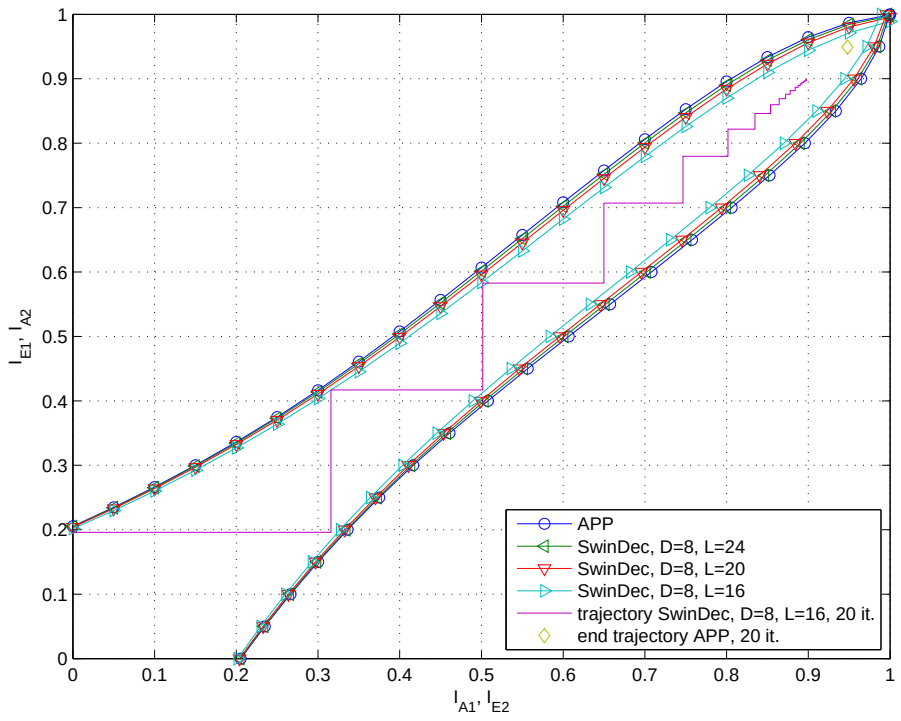



**Figure 5.30**: EXIT chart analysis of the (1932,640) UMTS turbo code decoded with different SwinDec component decoders at a $E_b/N_0$ value of 0.5 dB.

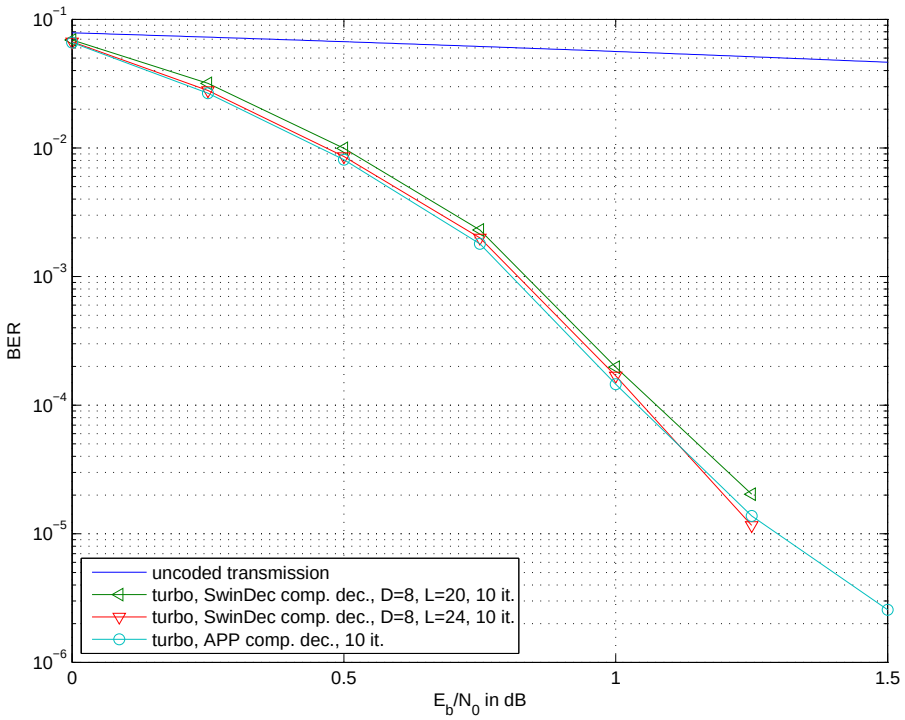



**Figure 5.31**: Decoding of the (1932,640) UMTS turbo code with different SwinDec component decoder configurations.

to an increased BER of the turbo decoder. Note that the characteristic curves for the different component decoders exhibit almost identical $I_E$ values for $I_A = 0$. This indicates that the component decoders will yield almost identical BER performance when no a priori information is available. The simulated trajectory for a sliding window turbo decoder using SwinDec component decoders with parameters $D = 8$, $L = 16$ and 20 iterations is also added in Fig. 5.30. The end of the corresponding trajectory for a turbo decoder with APP component decoders is also included as reference. This point is closer to the upper right corner ($I_A = 1, I_E = 1$) and thus indicates a lower BER. The results of the corresponding BER simulations for the sliding window turbo decoder are shown in Fig. 5.31 for 10 iterations. Here, a SwinDec decoder configuration with $D = 8$ and $L = 24$ closely approaches the performance of a turbo decoder with APP component decoders.

We now turn our focus to the effect of quantization of soft-information on the performance of sliding window turbo decoders. This is because of two reasons. First, in most practical applications the analog turbo decoder is preceded by some digital signal processing. It is then required to convert the available quantized soft information into analog inputs for the decoder. The second reason is that there are internal storage elements in the sliding window decoder which may optionally be implemented in the digital domain. Clearly, the complexity of the required D/A and A/D conversions heavily depends on the number of bits used for the quantization. In the following we assume the non-linear and SNR dependent quantization as described in Section 4.6 and investigate the impact on decoder performance for the example of the above UMTS turbo code. We start with an EXIT chart analysis as depicted in Fig. 5.32 for a $E_b/N_0$ value of 0.5 dB. The characteristic curves of an APP component decoder with 3 bit and 4 bit channel quantization are shown together with the corresponding curves in case both the channel information and the internal extrinsic information are quantized with 3 bit and 4 bit. The quantization of internal information is performed based on the channel SNR and not the inter-

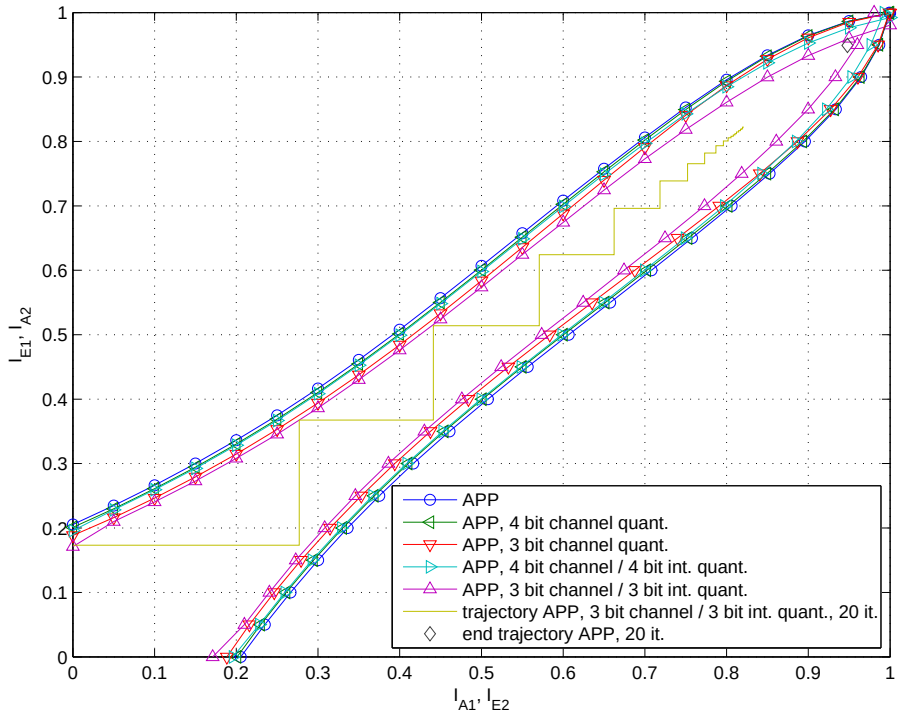



**Figure 5.32**: EXIT chart analysis of the (1932,640) UMTS turbo code decoded with different component decoders and quantization of soft information at a $E_b/N_0$ value of 0.5 dB.

nal SNR in the turbo decoder.[4] In case of quantization the characteristic curves appear to be shifted towards lower $I_E$ values. This effect again narrows the available tunnel for the trajectory of the turbo decoder which then causes an increased BER of the turbo decoder. Quantization leads to smaller $I_E$ values, even for $I_E = 0$. This in contrast to Fig. 5.30 and implies that the BER performance of the component decoders is also degraded. The simulated trajectory for a turbo decoder with 3 bit channel quantization, 3 bit internal quantization and 20 iterations is also added in Fig. 5.32. The end of the corresponding trajectory for a turbo decoder with APP component decoders and no quantization is again included as reference. As expected, this point is closer to the upper right corner ($I_A = 1, I_E = 1$) than in case of quantization thus indicating a lower BER.

We now investigate the performance of a turbo decoder with SwinDec component decoders in conjunction with the quantization of soft information. This combines the effects investigated in Fig. 5.30 and Fig. 5.32. We assume a SwinDec component decoder with $D = 8$, $L = 24$ and four bit channel quantization as it is, e.g., provided by a digital input interface. The BER performance of such a sliding window turbo decoder is depicted in Fig. 5.33 for 10 decoder iterations. A decoder with four bit channel quantization and (ideal) analog storage elements for the (internal) extrinsic information is compared to a decoder where both channel information and extrinsic information are quantized with four bits, i.e., digital storage elements are used. We recognize a small performance degradation in comparison with the corresponding curve of a turbo decoder with APP component decoders and no quantization. The quantization of the channel values with four bit at the input of the SwinDec decoder configuration causes an offset of around 0.04 dB. The additional quantization of internal extrinsic information with four bit leads to an additional shift of 0.08 dB. The overall performance degradation compared to the reference decoder is here 0.12 dB.

[4]The use of the internal SNR may slightly improve the BER performance of the turbo decoder.

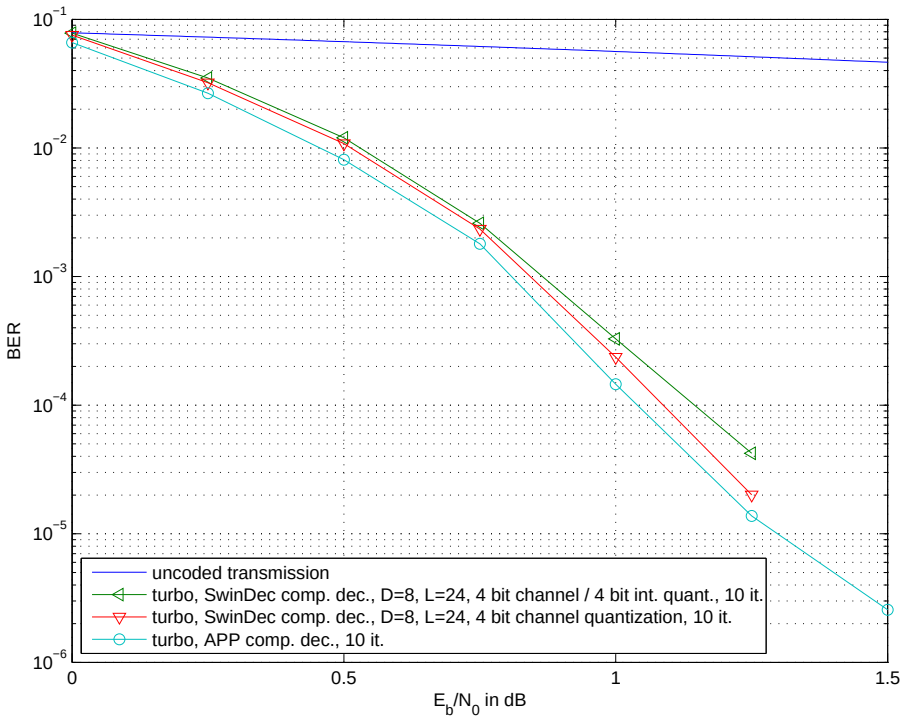



**Figure 5.33**: Decoding of the (1932,640) UMTS turbo code with different component decoders and 4 bit quantization.

The effect of quantization on the performance of a digital turbo decoder is investigated in, e.g., [WW99], [JH99], [MWW00] and [MW01].

A related sliding window turbo decoder where the time-continuous exchange of extrinsic information between the component decoders is partly maintained can be found in [ALS+05], [ALSJ05] [ASLJ06] and [Arz06]. Here, the sliding window technique is applied to the code graph of the overall turbo code rather than the component code.

### 5.4.3 Fully Parallel LDPC Decoder

LDPC codes are naturally suited for fully parallel decoder architectures. Typically, the block size of LDPC codes is rather large. This leads to fairly complex analog decoders which are difficult to simulate. In the following we use examples of rather short LDPC codes as they are proposed for the emerging IEEE standard 802.11n [IEE05]. We focus on the (648,324) LDPC code with rate $R = 1/2$ and the (648,540) LDPC code with rate $R = 5/6$. These codes can be represented by a normal graph based on the parity-check matrix as shown in Fig. 3.9. In order to obtain the normal graphs for the representation of the analog decoding networks we transform the normal graphs as outlined in Section 3.4.2. For this, we apply both the C3 Algorithm from Section 3.3.1 and the V3 Algorithm from Section 3.4.2 to the corresponding parity-check matrices of the codes. For the rate $R = 1/2$ and rate $R = 5/6$ codes we then obtain $2808 \times 3132$ and $3240 \times 3780$ expanded parity-check matrices, respectively, which utilize additional (internal) state variables. These matrices are simply referred to as $\boldsymbol{H}_3$, where the index reflects the degree of the nodes in the associated normal graph. The generic view of an analog LDPC decoder based on the transformed normal graph is depicted in Fig. 5.34. Note that every column $i$ in $\boldsymbol{H}$ with weight $d_{v,i}$ is split up into $d_{v,i} - 1$ equality node processors and that every row $j$ in $\boldsymbol{H}$ with weight $d_{c,j}$ is split up into $d_{c,j} - 2$ check node processors.

The performance of the fully parallel analog LDPC decoders is now evaluated with our

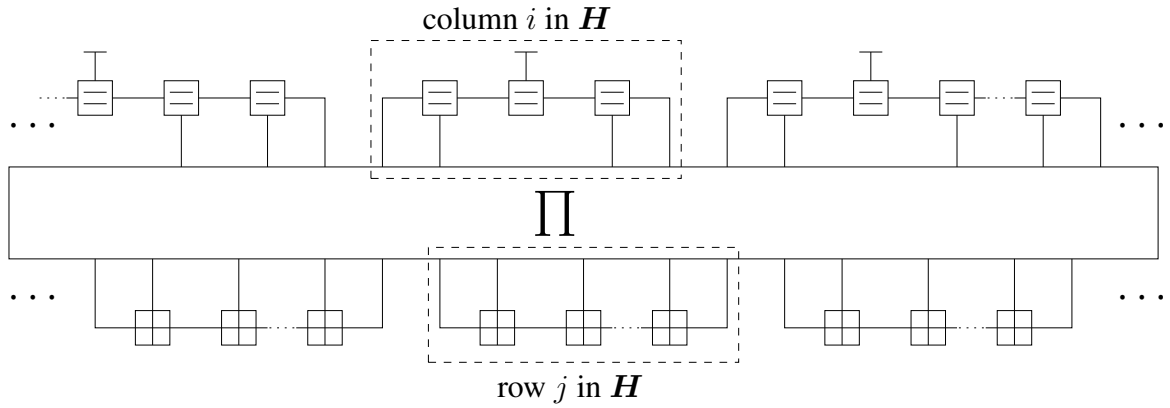



**Figure 5.34**: Generic view of an analog LDPC decoder.

different time-discrete simulation models. Note that for these codes our time-continuous decoder model in SIMULINK becomes too complex and time-consuming. Fig. 5.35 depicts the simulated BER for the rate $R = 1/2$ LDPC decoder as a function of the number of iterations. Three different simulation models are compared. The first simulation model represents conventional iterative decoding based on the original parity-check matrix of the code and the other two simulation models are based on the extended parity-check matrix $\boldsymbol{H}_3$ with $h = 1$ and $h = 0.1$, respectively. All simulation models were run on the same set of channel values in order to allow a fair comparison of the simulation results. Different $E_b/N_0$ values of 1.5 dB, 1.75 dB and 2 dB are depicted in Fig. 5.35. We recognize that decoding on the extended parity-check matrices $\boldsymbol{H}_3$ requires significantly more iterations in order to converge than decoding based on the original $\boldsymbol{H}$ matrix. Furthermore, when the step size $h$ in the simulation model is decreased from 1 to 0.1

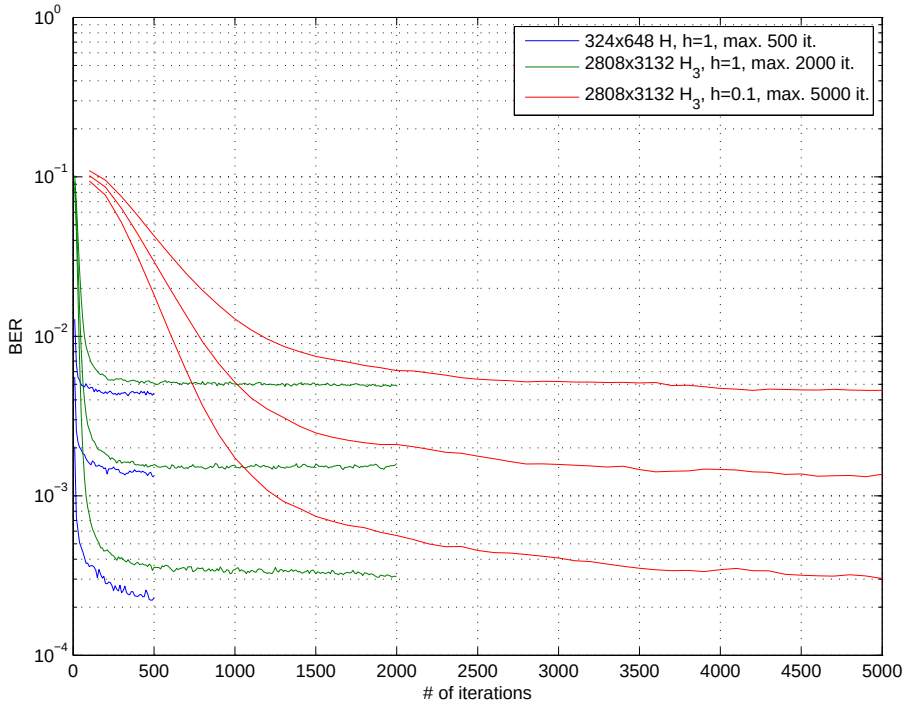



**Figure 5.35**: Simulated BER of different LDPC decoder simulation models for the (648,324) LDPC code at $E_b/N_0$ values of 1.5 dB, 1.75 dB and 2 dB.

the number of iterations needs to be increased accordingly. Here, it is important to emphasis that a larger number of iterations in the simulation model does not imply a longer settling time of the analog decoding network or increased complexity. More iterations simply originate from a finer time-resolution in the used simulation model. We can conclude from Fig. 5.35 that all three simulation models converge to quite comparable BER results for the different simulated SNR values. Neither the degree restriction on the nodes in the normal graph nor the step size of the simulation model significantly alters the BER performance. This behavior is somehow surprising since the decoders are based on two different code graphs and the message exchange in the decoding networks is also different. These results are in contrast to the results in [HB04] and [HB06] where a different BER performance is reported for different simulation models (and different codes).

In the following we further investigate the BER performance of fully parallel LDPC decoders for the (648,324) LDPC code with rate $R = 1/2$ and the (648,540) LDPC code with rate $R = 5/6$. For this, the decoders are evaluated with the time-discrete simulation model based on the extended parity-check matrix $\boldsymbol{H}_3$ with $h = 1$ and 250 iterations. These restrictions are necessary in order to limit the computational complexity. The corresponding simulation results are depicted in Fig. 5.36 together with an iterative reference decoder based on the original parity-check matrix $\boldsymbol{H}$ with 50 iterations. Note that both simulation models yield almost identical BER results for each of the two code rates.

In most practical applications the analog LDPC decoder is preceded by some digital signal processing. It is then required to convert the available quantized soft information into analog inputs for the decoder. The complexity of this conversion heavily depends on the number of bits per input sample. In the following we assume the non-linear and SNR dependent quantization as described in Section 4.6 and investigate the impact on the performance of the two LDPC decoders. For this, we use the time-discrete simulation model based on the $\boldsymbol{H}$ matrix of the codes, i.e., conventional iterative decoding, with 50 iterations. The simulation results for a

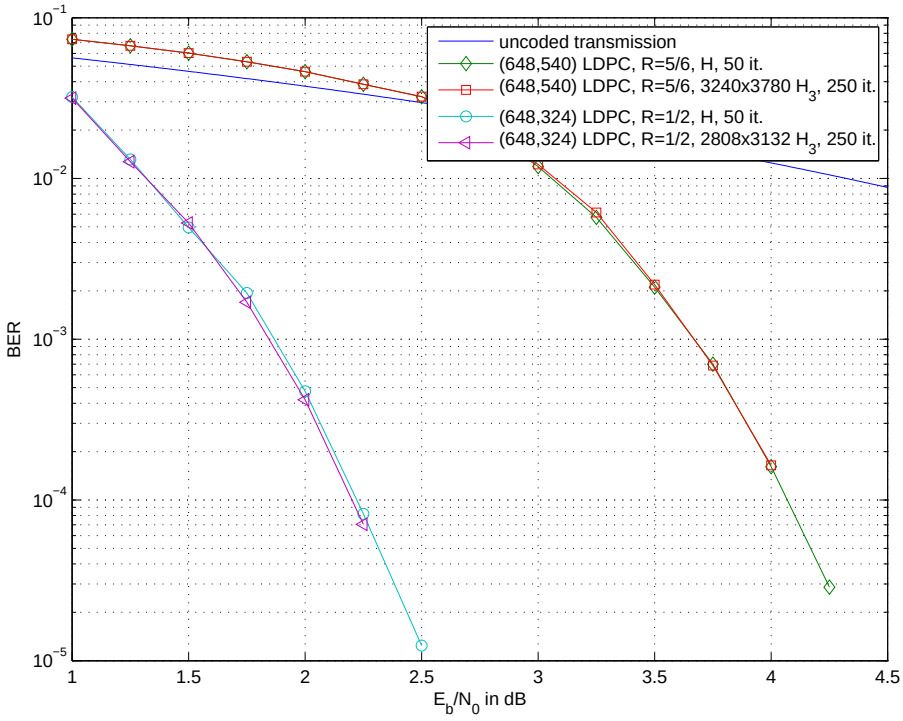



**Figure 5.36**: Decoding of the (648,324) LDPC code with rate $R = 1/2$ and the (648,540) LDPC code with rate $R = 5/6$ based on the corresponding $\boldsymbol{H}$ and $\boldsymbol{H}_3$ matrices.

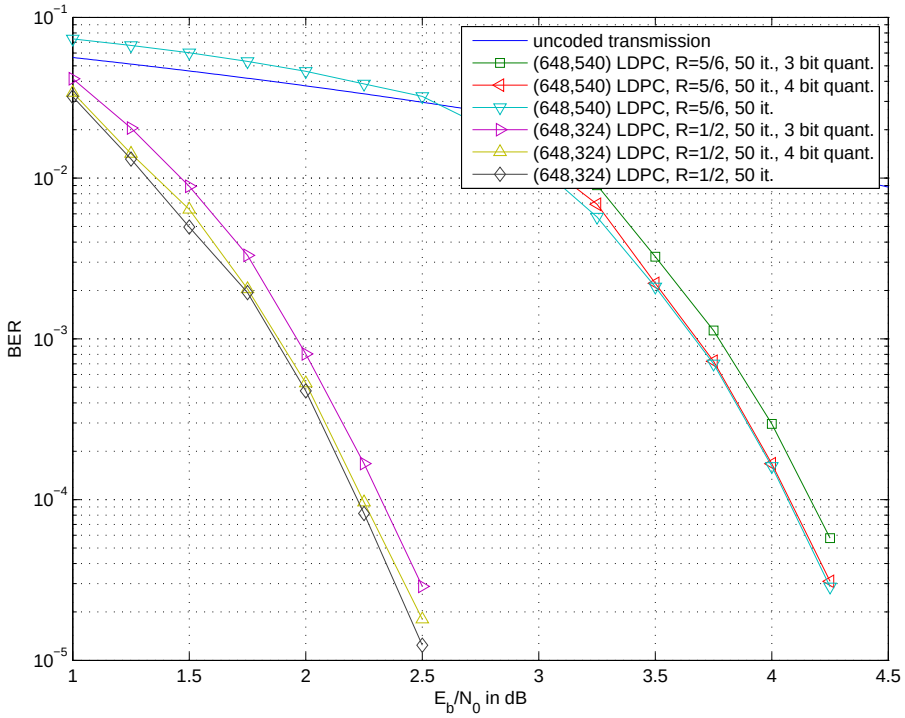



**Figure 5.37**: Decoding of the (648,324) LDPC code with rate $R = 1/2$ and the (648,540) LDPC code with rate $R = 5/6$ based on the $\boldsymbol{H}$ matrices with a quantization of the decoder input.

quantization of the decoder input with three and four bits are depicted in Fig. 5.37. With a quantization of three bit we observe a loss of 0.12 dB for the rate $R = 1/2$ code and a loss of 0.1 dB for the rate $R = 5/6$ code. The use of four bits reduces the loss in both cases to less than 0.05 dB. In this fully parallel decoder architecture the (internal) extrinsic information is inherently stored in the analog decoding network so that no internal quantization needs to be considered.

A realization of these analog LDPC decoders including digital input and output interfaces is further investigated in Chapter 8. The implementation of a simple analog LDPC decoder is reported in [HBP05] and [HBP06].

## 5.5 On the Possible Equivalence between Analog and Digital Decoding

The advent of analog decoding in [Hag97b], [Hag98], [HW98] and [LLHT98] gave rise to questions about the equivalence between analog and digital decoding. Analog decoding is implicitly understood as operating with analog, i.e., value-continuous, messages in continuous time. The analog decoder considered in this Chapter relies on the time-continuous simulation model from Section 5.1.1 where all node processors perform the ideal decoder operations. The speed of the analog decoder is only limited by the parasitics which are modeled by lumped RC elements. We furthermore assume that the output of the analog decoder is sampled after the decoding network has settled completely. In our context digital decoding performs the same ideal decoder operations but passes on the results in one full step to neighboring node processors. The message exchange is thus time-discrete and iterative. However, a digital message representation with a limited number of quantization levels is not considered. Hence, both analog and digital

decoders are simulated with a floating point message representation.

In this section we are more interested in a statistical evaluation of decoder performance in terms of the BER rather than proving equivalence in terms of identical soft outputs for all possible input configurations. In particular, we do not try to prove equivalence analytically. For many analog decoder configurations it is in fact easy to demonstrate that they are not equivalent to a conventional time-discrete decoder in a narrower sense. This is for example the case when analog and digital decoders are based on different code graphs.

In the following we differentiate between different types of code graphs which are considered in this work and summarize the results from the previous sections of this Chapter. The reader is also referred to the comments made on the equivalence between analog and digital decoding in [Sch05].

### 5.5.1 Code Graphs without Loops

Decoding networks based on code graphs without loops are easy to analyze. Possible examples include code graphs which form a tree or conventional trellis descriptions of block codes and terminated convolutional codes. Another example is the SwinDec decoder from Section 5.3 which implements only a small cycle-free sub-graphs of the overall code graph. The time-continuous message exchange on such code graphs does not alter the final decoder decision and the output of both analog and digital decoders is identical in both sign and magnitude.

### 5.5.2 Tailbiting Representations of Codes

Tailbiting representations of codes are a special case of code graphs with loops since there is only one typically large loop in the decoding network. We investigated tailbiting decoders for the (8,4,4) extended Hamming code in Section 5.2.2 and the (32,16) tailbiting convolutional code in Section 5.2.3. Our results suggest that analog tailbiting decoders yield identical BER results as the tailbiting wrap-around algorithm when the stabilization length is sufficiently large. The soft output may vary slightly depending on the chosen stabilization length in the digital decoder. The BER performance of the analog tailbiting decoder hereby closely approaches the BER performance of the APP decoder, even when the tailbiting loop is as small as four trellis sections as in case of the tailbiting representation of the (8,4,4) extended Hamming code. More comments on the equivalence between the tailbiting wrap-around decoder and the APP decoder can, e.g., be found in [AH98].

### 5.5.3 Code Graphs with Loops

Decoders based on code graphs with loops are more difficult to analyze. Here, we distinguish between decoders based on binary and non-binary code graphs. We start with binary code graphs as they are for example obtained from the parity-check matrix of the code. Analog and digital decoders based on such graphs are in general not equivalent in a narrower sense. This is because the normal graph needs to exhibit nodes with degree three in order to represent an analog decoding network. The normal graph thus typically needs to be transformed so that analog and digital decoders are based on essentially different code graphs. Examples for this are the (7,4,3) Hamming code and the (8,4,4) extended Hamming code[5] in Section 5.2.2 and the LDPC codes in Section 5.4.3. However, no evidence was found that this transformation of the code graph impacts decoder performance in terms of the BER. Other modifications of the normal graph, however, may improve the performance of both analog and digital decoders. This was also demonstrated in Section 5.2.2 for the example of the (7,4,3) Hamming code and the (8,4,4) extended Hamming code. It is particularly interesting to see that the modified

[5]The results are not plotted.

decoder for the (7,4,3) Hamming code closely approaches the BER performance of the APP decoder despite the presence of small cycles, see Fig. 5.10. For a given normal graph with degree three we found that different time-continuous and time-discrete simulation models yield almost identical BER results. Simulation results for this can be found in Section 5.2.2 and in Section 5.4.3. At this point it needs to be emphasized that different time-discrete simulation models require a significantly different number of iterations for convergence, see in particular Fig. 5.35. If this is not considered carefully one may be tempted to argue that one simulation model or decoder implementation is better than the other.

We now turn our focus to decoders based on non-binary code graphs with loops. A typical example of such codes are turbo codes. Turbo codes naturally lead to degree three nodes in the normal graph, see Fig. 4.9, so that there are no modifications of the code graph required. In Section 5.4.1 we presented examples of a (192,64) turbo code and a (192,96) turbo code as standardized for DVB [ETS03]. We found that a conventional iterative turbo decoder approaches the BER performance of a fully parallel analog turbo decoder with increasing number of iterations. These results are somehow surprising since the timing of the message exchange in the decoder is completely different. In case the turbo code is decoded with the analog sliding window turbo decoder from Section 5.4.2 the performance is equivalent to a turbo decoder which uses digital sliding window decoders with the same stabilization length.

Note that decoders based on code graphs with loops in general tend to overestimate the reliability of the decoder output. However, this effect is not further investigated since both analog and digital decoders are affected.

It is important to emphasis that digital decoders are based on a message representation with a finite number of quantization levels. This includes the channel values, the extrinsic information as well as all decoder internal messages. It is furthermore common practice to approximate the computations in the node processors in order to lower decoder complexity, see, e.g., [RVH95], [Daw96]. A digital decoder realization will thus not achieve the simulated BER performance of the ideal digital decoder assumed throughout this work. A carefully designed analog decoder can therefore be expected to outperform a digital decoder in terms of the BER.

# 6

# Integrated Circuits for Analog Decoding

There is a happy match between the various different message representations used in decoding algorithms and voltages or currents in analog integrated circuits. During the course of this chapter we demonstrate that a probability $P(x = i)$ is naturally represented by a normalized current $I_i/I_b$, with $I_b$ as sum current. The soft bit $\lambda(X)$ is then determined accordingly as the normalized differential current $\Delta I/I_b$. Similarly, normalized voltages represent messages in the logarithmic domain. Here, a single-ended voltage $V_i$ normalized by a constant $V_T$ can be interpreted as log-likelihood with offset $\psi$ and inverted sign. A normalized differential voltage $\Delta V/V_T$ then corresponds to a log-likelihood ratio $L(X)$. A summary of these relationships is given in Table 6.1. The equivalence between message representations and normalized voltages

**Table 6.1**: Happy match between different message representations and normalized voltages or currents.

| message representation | short notation | range | circuit quantity | short notation |
|---|---|---|---|---|
| probability | $P(x = i)$ | $0 \ldots 1$ | norm. current | $I_i/I_b$ |
| soft bit | $\lambda(X)$ | $-1 \ldots +1$ | norm. diff. current | $\Delta I/I_b$ |
| log-likelihood+const. | $l_i(X) + \psi$ | $-\infty \ldots 0$ | norm. voltage | $-V_i/V_T$ |
| log-likelihood ratio | $L(X)$ | $-\infty \ldots +\infty$ | norm. diff. voltage | $\Delta V/V_T$ |

and currents is based on an exponential characteristic of bipolar transistor circuits. For many applications it may be desirable to use complementary metal oxide semiconductor (CMOS) transistors rather than bipolar transistors. In general, CMOS devices exhibit a different characteristic which only roughly approximates the desired exponential behavior. We therefore use bipolar transistors in the following derivation of different decoder building blocks.

In Section 6.1 we introduce some elementary transistor circuits including relevant transistor models and derive the main building blocks for check node and variable node processors. We then generalize these building blocks to the case of non-binary node processors with non-binary message representations in Section 6.2. The interfacing circuits which are required in order to interconnect different building blocks are covered in Section 6.3. In Section 6.4 we give some examples of complete node processors and analyze the complexity of different decoder architectures for turbo codes and LDPC codes.

## 6.1 Elementary Transistor Circuits

This section introduces some elementary transistor circuits used in our analog decoding networks. This, for example, includes the differential pair and the pair of diode-connected transistors. The differential pair transforms the differential input voltage, interpreted as a L-value, into two output currents interpreted as the associated probabilities. The natural inverse to this circuit is the pair of diode-connected transistors. It transforms two input currents, interpreted as probabilities, into a differential output voltage representing the corresponding L-value. We demonstrate that a stacked configuration of differential pairs yields a probability multiplier and that different wiring networks at the output of such a multiplier lead to different decoder building blocks [MGYH00]. We start with the basic description of bipolar and CMOS transistor models and derive step-by-step the circuit implementations of check node and variable node processors. These transistor circuits are heavily used in any analog decoder implementation based on binary code graphs.

### 6.1.1 Transistor Models

A bipolar npn transistor is depicted in Fig. 6.1 together with the corresponding sign convention. The three terminals are denoted with collector (C), base (B) and emitter (E), respectively. This

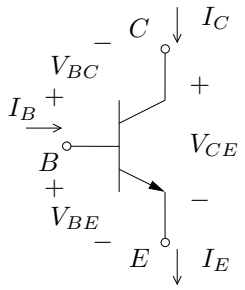



**Figure 6.1**: A bipolar npn transistor with sign convention.

transistor is biased to operate in the forward-active region, i.e., with a forward-biased B-E junction and a reverse-biased C-B junction. This mode of operation is typical for most applications and will be assumed throughout this work. The *Ebers-Moll* DC (large signal) model describes the relationship between the collector current $I_C$ and the B-E voltage $V_{BE}$ as

$$I_C = I_S \left( e^{\frac{V_{BE}}{V_T}} - 1 \right), \tag{6.1}$$

with $I_S$ as constant to describe the transfer characteristic of the transistor. Typical values of $I_S$ are $10^{-14}$ to $10^{-16}$ A [GM93]. $V_T$ is the thermal voltage, which is equal to $kT/q$ with $k$ being Boltzmann's constant ($k = 1.3807 \times 10^{-23}$ VAs/°K), $T$ the absolute temperature in degrees Kelvin and $q$ the charge of an electron ($q = 1.6 \times 10^{-19}$ As). At room temperature ($T = 300°$ K) we obtain $V_T \approx 26$ mV. For $V_{BE} \gg V_T$, (6.1) reduces to

$$I_C = I_S \, e^{\frac{V_{BE}}{V_T}}. \tag{6.2}$$

The base current $I_B$ is related to the collector current $I_C$ according to

$$I_B = \frac{I_C}{\beta_F} = \frac{I_S}{\beta_F} \, e^{\frac{V_{BE}}{V_T}}, \tag{6.3}$$

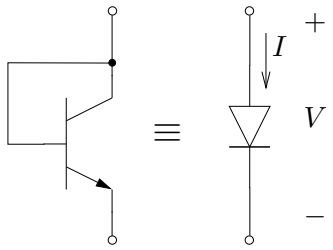



**Figure 6.2**: The diode-connected npn transistor.

where $\beta_F$ is the forward current gain of the transistor. Typical values of $\beta_F$ for npn transistors are between $50$ and $500$ [GM93]. The emitter current is the sum of the base and collector currents, i.e.

$$I_E = I_C + I_B = I_C + \frac{I_C}{\beta_F} = \frac{I_C}{\alpha_F}, \tag{6.4}$$

with $\alpha_F = \frac{\beta_F}{1+\beta_F}$, which is typically between $0.98$ and $0.998$ for npn transistors. For transistors with high current gain it is common practice to use the approximation $\alpha_F \approx 1$. When base and collector terminals of a npn transistor are shorted as shown in Fig. 6.2, the transistor behaves like a single pn diode. In this case we can reformulate (6.2) and express the voltage drop across the pn junction as

$$V = V_T \ln\left(\frac{I}{I_S}\right). \tag{6.5}$$

Such a diode-connected transistor exhibits a logarithmic characteristic as it can be used for transforming a probability represented as input current $I$ into a log-likelihood (plus a constant) represented as voltage drop $V$ across the diode.

For many applications it is desirable to work with CMOS devices rather than bipolar transistors. The CMOS equivalent of a npn transistor is the nMOS transistor shown in Fig. 6.3. It has four terminals denoted with drain (D), gate (G), source (S) and bulk (B). Here, we need to differentiate between three different operating regions, namely weak inversion, moderate inversion and strong inversion [Tsi99]. In weak inversion (or sub-threshold mode of operation) the drain current $I_D$ is due to diffusion and is exponentially related to the gate-source voltage $V_{GS}$ similar to a bipolar transistor. This is, however, no longer the case in moderate or strong inversion. Strong inversion is characterized by $V_{GS} > V_{th}$ with $V_{th}$ as the process dependent threshold voltage of the transistor. In this region the drain current $I_D$ is due to drift and for saturated devices with $V_{DS} > V_{GS} - V_{th}$ it is approximately quadric in $V_{GS}$. A simplified and

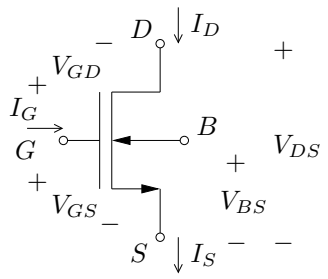



**Figure 6.3**: nMOS transistor with sign convention.

commonly used transistor model for this operation region is given by

$$I_D = \frac{1}{2}\,\mu\,\mathrm{C_{ox}}\,\frac{\mathrm{W}}{\mathrm{L}}\,(V_{GS} - V_{th})^2\,, \tag{6.6}$$

where $\mu$ denotes the surface mobility of the electrons and $\mathrm{C_{ox}}$ the oxide capacitance per unit area. The parameters W and L refer to the width and length of the channel, respectively. In moderate inversion the current is a result of of both drift and diffusion and is thus neither exponential nor polynomial. This region extends over several orders of magnitude in drain current and thus plays an important role in analog decoding. The fact that the behavior of CMOS devices for the important case of moderate inversion is neither exponential nor quadric makes it extremely difficult to analyze decoder performance analytically. In Section 8.3 we will evaluate the error of some decoder building blocks implemented with nMOS devices and estimate the expected performance when such blocks are used in large analog decoding networks.

Note that npn and nMOS transistors can always be replaced by their complementary pnp and pMOS transistors which exhibit a very similar characteristic. We then obtain complementary building blocks which can be used in, e.g., folded circuit architectures. In the following sections we use the more common npn transistors in order to introduce our transistor circuits. For a CMOS implementation of the decoder we simply substitute npn transistors with nMOS transistors.

### 6.1.2 Differential Pair

The differential pair (or differential amplifier) shown in the lower part of Fig. 6.4 is a widely used transistor configuration and plays a fundamental role in analog decoding. For matched transistors $Q_0$ and $Q_1$ we assume $\alpha_{F_0} = \alpha_{F_1} = \alpha_F$ and $I_{S_0} = I_{S_1} = I_S$. Hence,

$$I_b = I_{E_0} + I_{E_1} = \frac{I_{C_0}}{\alpha_F} + \frac{I_{C_1}}{\alpha_F}. \tag{6.7}$$

Using (6.2), the two output currents $I_{C_0}$ and $I_{C_1}$ can be expressed in terms of the two input voltages $V_0$ and $V_1$ according to

$$I_{C_0} = I_S\, e^{\frac{V_0 - V}{V_T}} \tag{6.8}$$

and

$$I_{C_1} = I_S\, e^{\frac{V_1 - V}{V_T}}, \tag{6.9}$$

with $V$ as the voltage at the common emitter. Equation (6.7) may now be written as

$$I_b = \frac{I_S}{\alpha_F}\, e^{-\frac{V}{V_T}} \left( e^{\frac{V_0}{V_T}} + e^{\frac{V_1}{V_T}} \right). \tag{6.10}$$

Solving for $e^{-\frac{V}{V_T}}$ yields

$$e^{-\frac{V}{V_T}} = \alpha_F \frac{I_b}{I_S}\, \frac{1}{e^{\frac{V_0}{V_T}} + e^{\frac{V_1}{V_T}}}. \tag{6.11}$$

When we substitute this expression in (6.8) and (6.9) we obtain for the output currents

$$I_{C_0} = I_S\, e^{\frac{V_0}{V_T}} e^{-\frac{V}{V_T}} = \alpha_F I_b \frac{e^{\frac{V_0}{V_T}}}{e^{\frac{V_0}{V_T}} + e^{\frac{V_1}{V_T}}} = \alpha_F \frac{I_b}{1 + e^{-\frac{V_0 - V_1}{V_T}}}, \tag{6.12}$$

and

$$I_{C_1} = I_S\, e^{\frac{V_1}{V_T}} e^{-\frac{V}{V_T}} = \alpha_F I_b \frac{e^{\frac{V_1}{V_T}}}{e^{\frac{V_0}{V_T}} + e^{\frac{V_1}{V_T}}} = \alpha_F \frac{I_b}{1 + e^{\frac{V_0 - V_1}{V_T}}}. \tag{6.13}$$

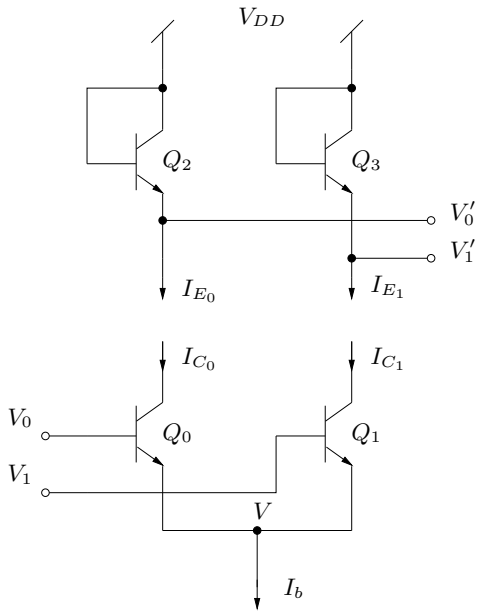



**Figure 6.4**: Differential pair (bottom) and pair of diode-connected transistors (top).

With $\Delta V = V_0 - V_1$ we obtain

$$I_{C_0} = \alpha_F \frac{I_b}{1 + e^{-\frac{\Delta V}{V_T}}} \tag{6.14}$$

and

$$I_{C_1} = \alpha_F \frac{I_b}{1 + e^{+\frac{\Delta V}{V_T}}}. \tag{6.15}$$

When $\Delta V$ is larger than about $200$ mV transistor $Q_0$ is almost completely switched on and $Q_1$ is almost completely switched off, i.e., $I_{C_0} \approx I_b$ and $I_{C_1} \approx 0$. With $\Delta V$ less than about $-200$ mV we observe the opposite behavior and $Q_1$ is switched on and $Q_0$ is switched off. The difference between the two output currents $I_{C_0}$ and $I_{C_1}$ is given as

$$\begin{aligned} \Delta I_C = I_{C_0} - I_{C_1} &= \alpha_F \frac{I_b}{1 + e^{-\frac{\Delta V}{V_T}}} - \alpha_F \frac{I_b}{1 + e^{+\frac{\Delta V}{V_T}}} \\ &= \alpha_F I_b \frac{e^{\frac{\Delta V}{V_T}} - 1}{e^{\frac{\Delta V}{V_T}} + 1}. \end{aligned} \tag{6.16}$$

Using the identity $\tanh\left(\frac{u}{2}\right) = \frac{e^u - 1}{e^u + 1}$, this equation results in

$$\Delta I_C = \alpha_F I_b \tanh\left(\frac{\Delta V}{2V_T}\right). \tag{6.17}$$

It unfolds in (6.17) that the differential pair provides a very efficient implementation of the hyperbolic tangent which is heavily used in decoding based on binary graphs. In the following we always assume transistors with high current gain so that $\alpha_F = 1$ is a reasonable good approximation. We then obtain from (6.14) and (6.15)

$$I_{C_0} = \frac{I_b}{1 + e^{-\frac{\Delta V}{V_T}}}$$

and

$$I_{C_1} = \frac{I_b}{1 + e^{+\frac{\Delta V}{V_T}}},$$

respectively. When we compare these equations with the message representations for binary random variables in Section 4.3.1 we can conclude that

$$I_{C_0} = I_b \, P\,(x = 0) \tag{6.18}$$

and

$$I_{C_1} = I_b \, P\,(x = 1)\,, \tag{6.19}$$

where

$$\Delta V = V_T \, L(X). \tag{6.20}$$

This reveals that the normalized output currents $I_{C_0}/I_b$ and $I_{C_1}/I_b$ represent the corresponding probabilities for bit $X$ when the normalized differential input voltage $\Delta V/V_T$ represents the corresponding L-value. The differential amplifier can therefore be used for the conversion of L-values into the corresponding probabilities. We can further conclude that the normalized differential output current $\Delta I_C/I_b$ in (6.17) represents the associated soft bit $\lambda(X)$. Also, the normalized single-ended input voltages $V_0/V_T$ and $V_1/V_T$ can be interpreted as the associated log-likelihoods $l_0(X)$ and $l_1(X)$ plus an additive constant.

### 6.1.3 Pair of Diode-Connected Transistors

The pair of diode-connected transistors shown in the upper part of Fig. 6.4 forms the natural inverse to the differential pair described in the last section. Here, the single-ended output voltages can be expressed as

$$V_0' = V_{DD} - V_T \ln\left(\frac{I_{E_0}}{I_S}\right) \tag{6.21}$$

and

$$V_1' = V_{DD} - V_T \ln\left(\frac{I_{E_1}}{I_S}\right), \tag{6.22}$$

with $V_{DD}$ as supply voltage. Similar to the differential pair we can interpret the two normalized input currents $I_{E_0}/I_b$ and $I_{E_1}/I_b$ as the two probabilities $P(x = 0)$ and $P(x = 1)$, respectively. When the voltage drops $V_{DD} - V_0'$ and $V_{DD} - V_1'$ generated across the two diode connected transistors are normalized with $V_T$ we obtain the corresponding log-likelihoods $l_0(X)$ and $l_1(X)$ plus an additive constant. With $I_{E_0} = I_b/2 + \Delta I_C/2$ and $I_{E_1} = I_b/2 - \Delta I_C/2$ we obtain for the differential output voltage

$$\begin{aligned} \Delta V' = V_1' - V_0' &= -V_T \ln\left(\frac{I_b/2 - \Delta I_C/2}{I_S}\right) + V_T \ln\left(\frac{I_b/2 + \Delta I_C/2}{I_S}\right) && (6.23) \\ &= V_T \ln\left(\frac{I_b/2 + \Delta I_C/2}{I_b/2 - \Delta I_C/2}\right) = V_T \ln\left(\frac{1 + \frac{\Delta I_C}{I_b}}{1 - \frac{\Delta I_C}{I_b}}\right). && (6.24) \end{aligned}$$

Note that the differential output voltage $\Delta V'$ in (6.23) is defined differently compared to the differential input voltage of the differential pair in the last section, i.e., a sign inversion occurs. Using the identity $\tanh^{-1}(u) = \frac{1}{2} \ln\left(\frac{1+u}{1-u}\right)$ equation (6.24) yields

$$\Delta V' = 2V_T \tanh^{-1}\left(\frac{\Delta I_C}{I_b}\right). \tag{6.25}$$

Like the differential pair provides an efficient means of implementing the hyperbolic tangent, this circuit is an efficient circuit implementation for the inverse hyperbolic tangent. We can prove the duality of the two transistor circuits by directly connecting the current outputs of the

differential pair with the current inputs of the pair of diode-connected transistors in Fig. 6.4. With (6.17) and (6.24) we obtain

$$\Delta V' = 2V_T \tanh^{-1}\left(\tanh\left(\frac{\Delta V}{2V_T}\right)\right) = \Delta V. \tag{6.26}$$

It unfolds that the normalized differential output voltage $\Delta V'/V_T$ again represents $L(X)$. This connected circuit configuration thus performs the conversion of an L-value into the corresponding probabilities and then back again from the probabilities to the L-value.

In the following sections we use the differential pair and the pair of diode connected transistors in order to design some elementary building blocks for decoders based on binary code graphs.

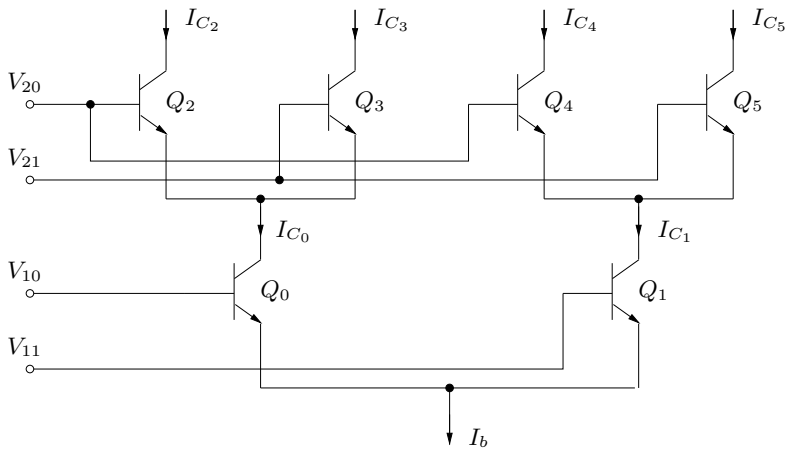



**Figure 6.5**: Stacked configuration of differential pairs.

### 6.1.4 Stacked Configuration of Differential Pairs

The last two sections covered the differential pair and the pair of diode-connected transistors. These circuits can be employed in order to transform L-values into the corresponding probabilities and vice versa. In this section, we introduce the stacked configuration of differential pairs which not only transforms one signal representation into another, but also features multiplication and facilitates extremely low complexity summation of probabilities as we will see in the following. A stacked configuration of differential pairs is depicted in Fig. 6.5. Here, the output currents $I_{C_0}$ and $I_{C_1}$ of the lower differential pair $(Q_0, Q_1)$ with the differential input voltage $\Delta V_1 = V_{10} - V_{11}$ provide the biasing currents for the upper two differential pairs $(Q_2, Q_3)$ and $(Q_4, Q_5)$. A second differential input voltage $\Delta V_2 = V_{20} - V_{21}$ is applied concurrently to the upper two differential pairs as shown in Fig. 6.5. We assume that all devices are matched, i.e., have identical $\alpha_F$ and $I_S$. The four output currents of this stacked configuration can now be

expressed in terms of the two differential input voltages as

$$I_{C_2} = \frac{I_{C_0}}{1+e^{-\frac{\Delta V_2}{V_T}}} = \frac{I_b}{\left(1+e^{-\frac{\Delta V_1}{V_T}}\right)\left(1+e^{-\frac{\Delta V_2}{V_T}}\right)}, \tag{6.27}$$

$$I_{C_3} = \frac{I_{C_0}}{1+e^{+\frac{\Delta V_2}{V_T}}} = \frac{I_b}{\left(1+e^{-\frac{\Delta V_1}{V_T}}\right)\left(1+e^{+\frac{\Delta V_2}{V_T}}\right)}, \tag{6.28}$$

$$I_{C_4} = \frac{I_{C_1}}{1+e^{-\frac{\Delta V_2}{V_T}}} = \frac{I_b}{\left(1+e^{+\frac{\Delta V_1}{V_T}}\right)\left(1+e^{-\frac{\Delta V_2}{V_T}}\right)}, \tag{6.29}$$

$$I_{C_5} = \frac{I_{C_1}}{1+e^{+\frac{\Delta V_2}{V_T}}} = \frac{I_b}{\left(1+e^{+\frac{\Delta V_1}{V_T}}\right)\left(1+e^{+\frac{\Delta V_2}{V_T}}\right)}. \tag{6.30}$$

Again, we interpret the normalized differential input voltages $\Delta V_1/V_T$ and $\Delta V_2/V_T$ as $L(X_1)$ and $L(X_2)$, respectively. Based on Section 6.1.2 and (6.27) to (6.30) we can then conclude that the normalized output currents satisfy

$$\frac{I_{C_0}}{I_b} = P(x_1=0), \tag{6.31}$$

$$\frac{I_{C_1}}{I_b} = P(x_1=1), \tag{6.32}$$

$$\frac{I_{C_2}}{I_b} = P(x_1=0)P(x_2=0), \tag{6.33}$$

$$\frac{I_{C_3}}{I_b} = P(x_1=0)P(x_2=1), \tag{6.34}$$

$$\frac{I_{C_4}}{I_b} = P(x_1=1)P(x_2=0), \tag{6.35}$$

$$\frac{I_{C_5}}{I_b} = P(x_1=1)P(x_2=1). \tag{6.36}$$

It becomes apparent that the stacked configuration of differential pairs transforms the input L-values into the corresponding probabilities and then performs a pairwise multiplication of them. The four possible probability products are then available as normalized output currents. The current output of this circuit allows a simple summation of the results according to Kirchhoff's current law. Depending on the required function of the circuit we can simply connect output wires together. Two particularly important wiring configurations are covered in the following sections.

### 6.1.5 Boxplus Circuit

Our first analog decoder building block is the Boxplus circuit which performs the operations of a degree three check node processor. It is based on the well-known Gilbert multiplier [Gil68], [GM93] and relies on a stacked configuration of differential pairs as introduced in the previous section. The Gilbert multiplier is obtained when the four outputs are connected together as shown in Fig. 6.6. The two output currents of the Gilbert multiplier are then given as $I_{C_{25}} = I_{C_2} + I_{C_5}$ and $I_{C_{34}} = I_{C_3} + I_{C_4}$. With (6.33) to (6.36) we obtain for the two normalized output currents of the Gilbert multiplier

$$\frac{I_{C_{25}}}{I_b} = P(x_1=0)P(x_2=0) + P(x_1=1)P(x_2=1), \tag{6.37}$$

$$\frac{I_{C_{34}}}{I_b} = P(x_1=1)P(x_2=0) + P(x_1=0)P(x_2=1). \tag{6.38}$$

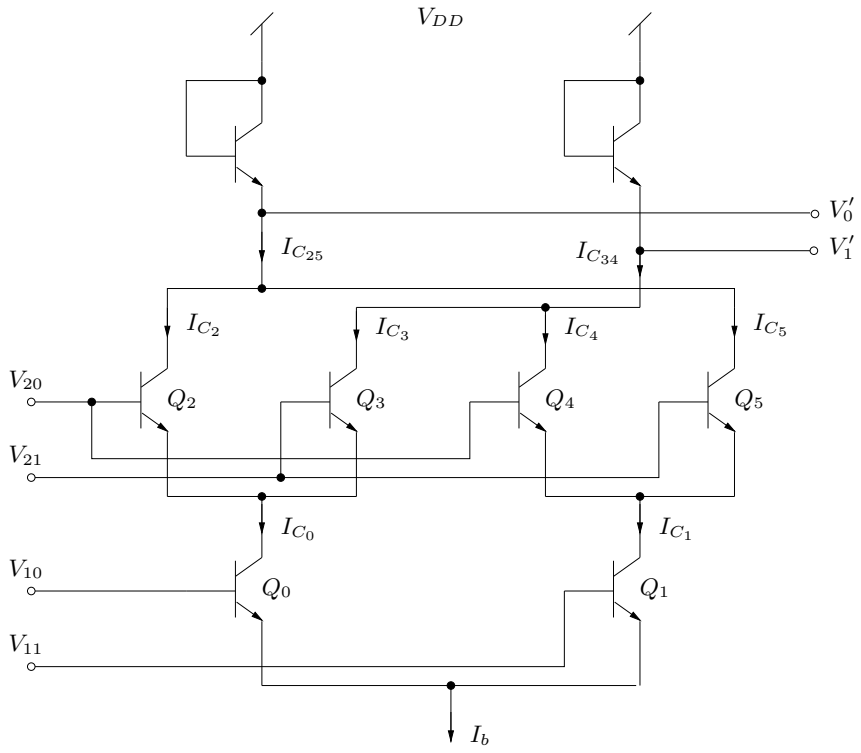



**Figure 6.6**: The Boxplus circuit consisting of a Gilbert multiplier and a pair of diode-connected transistors stacked on top.

This proves that this circuit configuration performs the basic operations of a check node processor with degree three, see Fig. 4.2. The normalized differential output current is given as

$$\begin{aligned}\frac{\Delta I_C}{I_b} &= \frac{I_{C_{25}} - I_{C_{34}}}{I_b} = \frac{1 + e^{\frac{\Delta V_1 + \Delta V_2}{V_T}} - e^{+\frac{\Delta V_1}{V_T}} - e^{+\frac{\Delta V_2}{V_T}}}{\left(1 + e^{+\frac{\Delta V_1}{V_T}}\right)\left(1 + e^{+\frac{\Delta V_2}{V_T}}\right)} \\ &= \frac{\left(1 - e^{+\frac{\Delta V_1}{V_T}}\right)\left(1 - e^{+\frac{\Delta V_2}{V_T}}\right)}{\left(1 + e^{+\frac{\Delta V_1}{V_T}}\right)\left(1 + e^{+\frac{\Delta V_2}{V_T}}\right)}. \end{aligned} \tag{6.39}$$

Using the identity $\tanh\left(\frac{u}{2}\right) = \frac{e^u - 1}{e^u + 1}$ we yield the well-known description of the Gilbert multiplier

$$\frac{\Delta I_C}{I_b} = \tanh\left(\frac{\Delta V_1}{2V_T}\right) \tanh\left(\frac{\Delta V_2}{2V_T}\right). \tag{6.40}$$

Provided that the two differential input voltages are small enough, we can use the linear approximation of the hyperbolic tangent. In this case, the Gilbert multiplier performs the linear four-quadrant analog multiplication according to

$$\frac{\Delta I_C}{I_b} \approx \left(\frac{\Delta V_1}{2V_T}\right)\left(\frac{\Delta V_2}{2V_T}\right). \tag{6.41}$$

However, this approximation only holds for a very restricted range of the differential input voltages with $\Delta V_1$ and $\Delta V_2 \ll V_T$. There are solutions available in the literature to circumvent this limitation due to the inherent non-linearity [Gil68]. It is interesting to note that exactly this non-linearity of the circuit is exploited in analog decoding, thus leading to very area- and power-efficient decoder implementations.

For the Boxplus circuit diode-connected transistors are stacked on top of the two rows of differential pairs as shown in Fig. 6.6. The voltage drop across each diode-connected transistor is proportional to the natural logarithm of the current through the device, see (6.5). Stacking the pair of diode-connected transistors on top of the Gilbert multiplier and defining the current components $I = I_b/2$ and $\Delta I_C = (I_{C_{25}} - I_{C_{34}})/2$ yields for the normalized differential output voltage

$$\begin{aligned} \frac{\Delta V'}{V_T} = \frac{V_1' - V_0'}{V_T} &= 2\tanh^{-1}\left(\frac{I_{C_{25}} - I_{C_{34}}}{2}\frac{2}{I_b}\right) \\ &= 2\tanh^{-1}\left[\tanh\left(\frac{\Delta V_1}{2V_T}\right)\tanh\left(\frac{\Delta V_2}{2V_T}\right)\right]. \end{aligned} \tag{6.42}$$

This expression corresponds to a reformulation of the Boxplus equation in terms of differential voltages. With $\Delta V_1/V_T = L(X_1)$ and $\Delta V_2/V_T = L(X_2)$ we obtain $\Delta V'/V_T = L(X_3)$, where $L(X_3) = L(X_1) \boxplus L(X_2)$. Note that the definition of the differential output voltage in (6.42) as $\Delta V' = V_1' - V_0'$ is different compared to the definition of the differential input voltages $\Delta V_1 = V_{10} - V_{11}$ and $\Delta V_2 = V_{20} - V_{21}$. This is because the output reflects the difference of the single-ended output voltages (with respect to ground) rather than the difference of the voltage drops across the diodes. In the binary case we thus need to change the polarity of the differential output voltage as described in the above.

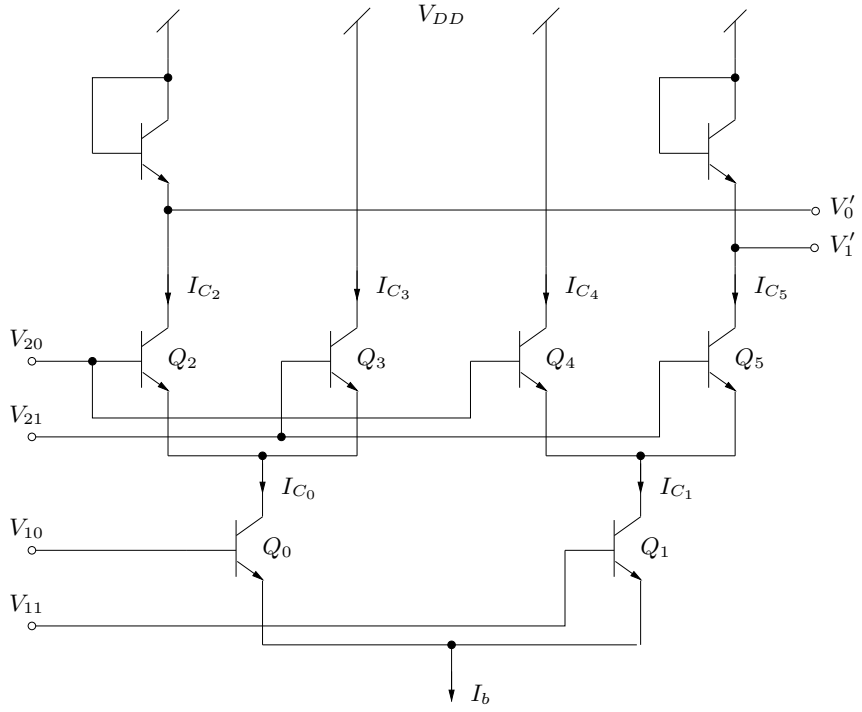



**Figure 6.7**: The summation circuit.

### 6.1.6 Summation Circuit

The second analog decoder building block is the summation circuit which performs the operations of a degree three variable node processor. Similar to the Boxplus circuit in Section 6.1.5 it is also relies on a stacked configuration of differential pairs with an additional pair of diode-connected transistors stacked on top. The only difference compared to the Boxplus circuit in Fig. 6.6 lies in the connectivity. When we apply the two currents $I_{C_2}$ and $I_{C_5}$ to the

diode-connected transistors as shown in Fig. 6.7 we obtain for the differential output voltage

$$\begin{aligned}\Delta V' = V_1' - V_0' &= V_T \ln\left(\frac{I_{C_2}}{I_S}\right) - V_T \ln\left(\frac{I_{C_5}}{I_S}\right) \\ &= V_T \ln\left(\frac{\left(1+e^{+\frac{\Delta V_1}{V_T}}\right)\left(1+e^{+\frac{\Delta V_2}{V_T}}\right)}{\left(1+e^{-\frac{\Delta V_1}{V_T}}\right)\left(1+e^{-\frac{\Delta V_2}{V_T}}\right)}\right) \\ &= \Delta V_1 + \Delta V_2. \end{aligned} \tag{6.43}$$

With $\Delta V_1/V_T = L(X_1)$, $\Delta V_2/V_T = L(X_2)$ and $\Delta V'/V_T = L(X_3)$ we observe the desired behavior $L(X_3) = L(X_1)+L(X_2)$. Note that not all output currents contribute to the calculation of the output voltage $\Delta V'$. Unused output currents are either connected directly to $V_{DD}$ as shown in Fig. 6.7 or through other diode-connected transistors which are used as dummy load.

A subtraction of L-values can be achieved when the output currents $I_{C_3}$ and $I_{C_4}$ are applied to the diode-connected transistors, or, when the polarity of the corresponding differential input voltage is changed. This can be realized by simply interchanging the associated input wires.

## 6.2 Generalized Building Blocks

The Boxplus and the summation circuits introduced in Sections 6.1.5 and 6.1.6 are the main building blocks for analog decoders based on binary code graphs where the messages are conveniently represented as L-values. In many cases the graph representation of a code is not binary and therefore requires more generalized building blocks. In the following we therefore extend the circuits introduced in Section 6.1 to non-binary code graphs with non-binary message representations. Based on the differential pair in Section 6.1.2 and the pair of diode connected transistors in Section 6.1.3 we obtain the probability multiplexor and the inverse probability multiplexor, respectively. A stacked configuration of such probability multiplexors then leads to a general probability multiplier circuit. In combination with some connectivity at its output and the inverse probability multiplexor stacked on top we obtain a general decoder building block which applies to arbitrary node processors. The basic principles of this section are already published in [MHO00].

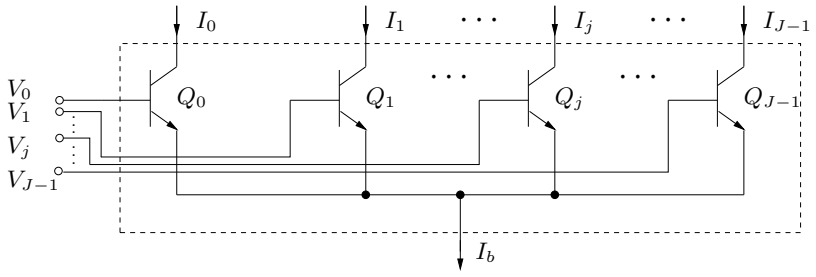



**Figure 6.8**: Probability multiplexor (pMUX) consisting of emitter coupled transistors.

### 6.2.1 Probability Multiplexor

The generalization of an emitter coupled pair to more than two transistors is depicted in Fig. 6.8. The arrangement of $J$ emitter coupled transistors $Q_j$ with $j \in \{0, \ldots, J-1\}$ allows the representation of a random variable $X$ with $J$ possible outcomes. The bias current $I_b$ is split up into

$J$ output currents $I_j$ depending on the set of single-ended input voltages $V_j$ with

$$I_j = I_b \, \frac{e^{\frac{V_j}{V_T}}}{\sum_{\nu=0}^{J-1} e^{\frac{V_\nu}{V_T}}}. \tag{6.44}$$

This expression is a straightforward generalization of (6.14) and (6.15). Following the results in (6.18) and (6.19) we can interpret the output currents $I_j$ as the corresponding probabilities weighted with the bias current $I_b$, i.e.

$$I_j = I_b \, P(x = j), \tag{6.45}$$

with $\sum_{j=0}^{J-1} I_b \, P(x = j) = I_b$. The circuit configuration in Fig. 6.8 is therefore also referred to as probability multiplexor (pMUX). The single-ended input voltages $V_j$ represent the log-likelihood values of the corresponding probabilities according to

$$V_j = l_j(X) \, V_T + \Psi, \tag{6.46}$$

with $\Psi$ as common DC voltage offset. The voltage $\Psi$ in (6.46) is chosen so that the appropriate voltage levels are provided at the input of the circuit without changing the differential characteristic. The differential input voltages $V_{i,j} = V_i - V_j$ in Fig. 6.8 represent the log-likelihood ratios of random variable $X$ according to

$$V_{i,j} = l_i(X) \, V_T - l_j(X) \, V_T = V_T \, L_{i,j}(X). \tag{6.47}$$

Note that the binary equivalent of (6.47) is stated in (6.20) with $\Delta V = V_{0,1}$.

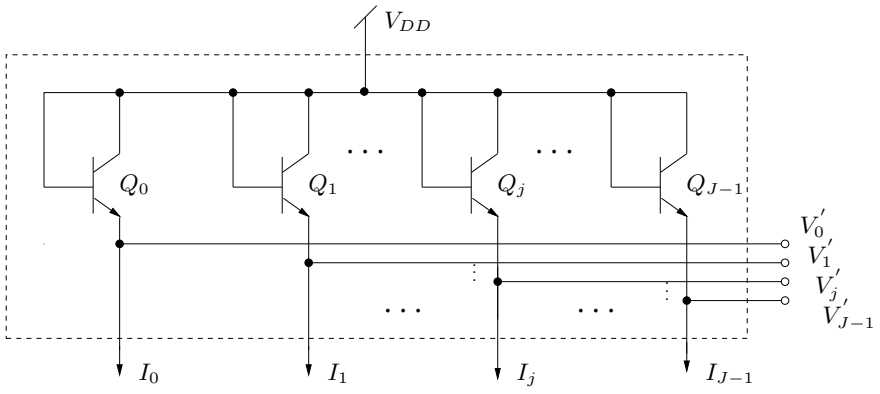



**Figure 6.9**: Inverse probability multiplexor (pMUX$^{-1}$) consisting of diode-connected transistors.

### 6.2.2 Inverse Probability Multiplexor

The arrangement of $J$ diode-connected transistors as shown in Fig. 6.9 is the natural inverse of the probability multiplexor in Fig. 6.8. This configuration is a generalization of the diode-connected transistor pair in Fig. 6.4 in order to represent random variables with $J$ possible outcomes. The single-ended output voltages $V_j'$ are determined by the input currents $I_j$ with $j \in \{0, \ldots, J-1\}$ and the supply voltage $V_{DD}$ according to

$$V_j' = V_{DD} - V_T \ln\left(\frac{I_j}{I_S}\right) \tag{6.48}$$

$$= -V_T \ln(I_j) + V_{DD} + V_T \ln(I_S). \tag{6.49}$$

We assume that the input currents $I_j$ represent the probabilities of a random variable $X$ as in (6.45). The single-ended output voltages $V_j'$ then represent the corresponding log-likelihoods according to

$$V_j' = -l_j(X)\ V_T + \Psi', \tag{6.50}$$

with $\Psi'$ as common DC voltage offset. Note the negative sign of $l_j(X)$ in (6.50). The differential output voltages $V_{i,j}' = V_i' - V_j'$ in Fig. 6.9 are then determined by

$$V_{i,j}' = l_j(X)\ V_T - l_i(X)\ V_T. \tag{6.51}$$

A comparison with (6.47) yields

$$V_{i,j}' = -V_{i,j} = -V_T\ L_{i,j}(X). \tag{6.52}$$

The circuit thus transforms probabilities represented as normalized input currents into the corresponding negative log-likelihoods plus an additive constant which are represented as normalized output voltages. The normalized differential output voltages then represent non-binary L-values similar to (6.47), but with inverted sign. It unfolds that the pair of diode-connected transistors in Fig. 6.9 essentially performs the inverse operation of the pMUX in Fig. 6.8. The circuit is thus also referred to as pMUX$^{-1}$.

In order to obtain the desired input for a consecutive stage we need to manipulate the voltage output of the pMUX$^{-1}$. For the binary case with $J = 2$ we only need to change the polarity of the differential output voltage by interchanging the output wires as described in Section 6.1.3. For $J > 2$ an additional inverter stage consisting of a pMUX and a pMUX$^{-1}$ becomes necessary as outlined in Section 6.4.

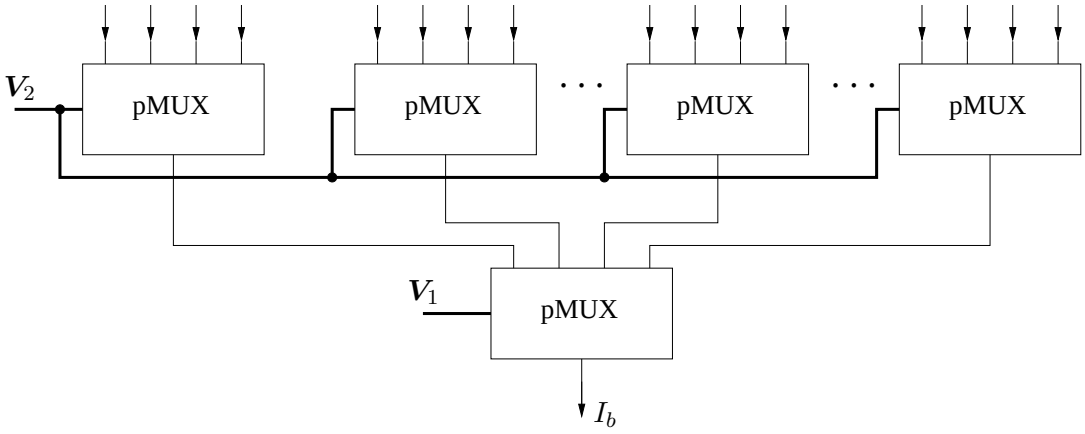



**Figure 6.10**: The stacked configuration of probability multiplexors for probability multiplication.

### 6.2.3 Generalized Multiplier Circuit

The probability multiplication of two in general non-binary random variables $X_1$ and $X_2$ can be achieved by a stacked configuration of probability multiplexors as shown in Fig. 6.10. The binary version of this multiplier is depicted in Fig. 6.5. Each output of the lower pMUX is connected to a pMUX in the upper row. We assume that the two random variables $X_1$ and $X_2$ have $A$ and $B$ possible outcomes, respectively. The inputs are then conveniently described by the voltage vectors

$$\boldsymbol{V}_1 = [V_{10}, V_{11}, \ldots, V_{1a}, \ldots, V_{1(A-1)}] \tag{6.53}$$

and

$$\boldsymbol{V}_2 = [V_{20}, V_{21}, \ldots, V_{2b}, \ldots, V_{2(B-1)}], \tag{6.54}$$

with $a \in \{0, \ldots, A-1\}$ and $b \in \{0, \ldots, B-1\}$. Vector $\boldsymbol{V}_1$ is applied to the pMUX in the lower row and $\boldsymbol{V}_2$ is applied concurrently to each pMUX in the upper row. For ease of exposition the routing of the vector signals is illustrated in Fig. 6.10 with thick lines. Using (6.46) we can interpret the two voltage vectors $\boldsymbol{V}_1$ and $\boldsymbol{V}_2$ as vectors of scaled log-likelihoods with the DC voltage offsets $\Psi_1$ and $\Psi_2$, respectively. Hence,

$$\boldsymbol{V}_1 = [(l_0(X_1)V_T + \Psi_1), (l_1(X_1)V_T + \Psi_1), \ldots, (l_a(X_1)V_T + \Psi_1), \ldots, (l_{A-1}(X_1)V_T + \Psi_1)] \tag{6.55}$$

and

$$\boldsymbol{V}_2 = [(l_0(X_2)V_T + \Psi_2), (l_1(X_2)V_T + \Psi_2), \ldots, (l_b(X_2)V_T + \Psi_2), \ldots, (l_{B-1}(X_2)V_T + \Psi_2)]. \tag{6.56}$$

The multiplier in Fig. 6.10 then calculates $C = AB$ output currents $I_c$ according to

$$I_c = I_b \frac{e^{\frac{V_{1a}}{V_T}}}{\sum_{\nu=0}^{A-1} e^{\frac{V_{1\nu}}{V_T}}} \frac{e^{\frac{V_{2b}}{V_T}}}{\sum_{\mu=0}^{B-1} e^{\frac{V_{2\mu}}{V_T}}}, \tag{6.57}$$

with $c = aB + b$ and $c \in \{0, \ldots, AB-1\}$. The expression in (6.57) is the straightforward generalization of (6.27) to (6.30) to non-binary random variables. We then find that the normalized output currents represent the products of the associated probabilities, i.e.

$$\frac{I_c}{I_b} = P(x_1 = a)P(x_2 = b), \tag{6.58}$$

which is the generalization of (6.33) to (6.36).

### 6.2.4 General Block Structure

The structure of our general decoder building block is depicted in Fig. 6.11. It relies on the stacked configuration of probability multiplexors (pMUX) for the multiplication of probabilities as introduced in the previous section. Again, we assume that the input voltage vectors $\boldsymbol{V}_1$ and $\boldsymbol{V}_2$ represent random variables with $A$ and $B$ possible outcomes, respectively, refer to (6.53), (6.54) and (6.55), (6.56). Similar to the binary building blocks in Sections 6.1.5 and 6.1.6 there is a connectivity block and an inverse probability multiplexor for the generation of the output voltages. The connectivity element simply consists of a wiring network which allows the summation of arbitrary output currents according to Kirchhoff's current law. Depending on the connectivity different probability products can be summed up in order to achieve the desired function of the building block. It is important to note that the stacked configuration of pMUX always calculates all $AB$ possible probability products. This is because each pMUX requires all signal components of the corresponding input vector for internal probability normalization. However, not all of the probability products may contribute to the output of the block. Unused outputs can directly be connected to $V_{DD}$ or through a diode-connected transistor used as dummy load. The $C'$, $C' \leq C$, current outputs of the connectivity element are applied to an inverse probability multiplexor (pMUX$^{-1}$) which transforms the input currents into $C'$ output voltages represented by vector $\boldsymbol{V}'$. The elements of $\boldsymbol{V}'$ have the form of (6.50). Note that the sum current in the pMUX$^{-1}$ may be smaller than $I_b$ which leads to a scaling of the output currents. However, this scaling only affects the probabilities and not the differential output voltages representing log-likelihood ratios. The general concept of stacking pMUX for probability multiplication can be further extended to stacks with more than two rows of pMUX, i.e., more then two input variables. This approach, however, is limited by the supply voltage.

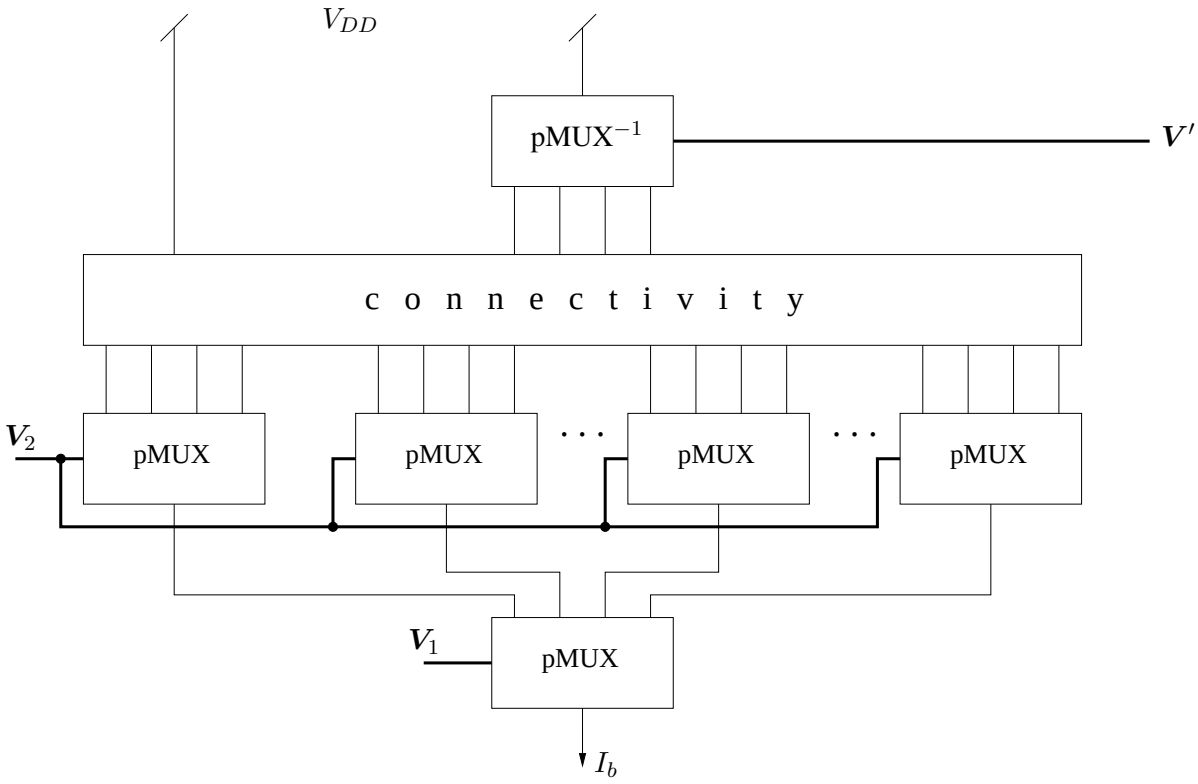



**Figure 6.11**: Structure of the general decoder building block.

The general decoder building block in Fig. 6.11 requires a total number of

$$A(1+B)+C', \tag{6.59}$$

transistors. In case the outputs of two or more transistors in a pMUX do not contribute to the output calculation the transistors can be replaced by a single transistor with dedicated input voltage [Win04]. However, this approach does not necessarily lead to a complexity reduction since additional circuity is required in order to generate this input. In this work we will make heavy use of the general decoder building block as shown in Fig. 6.11.

In the following we discuss different interfacing stages in order to connect the output of one building block to the input of another building block.

## 6.3 Interfacing between Blocks

Analog decoding networks consist of node processors which are interconnected according to the corresponding normal graph representation of the code. In order to connect the voltage output of one node processor to the voltage input of another node processor we typically utilize special output buffer stages. The purpose of such output buffer stages is twofold. First, the output voltage levels of a block need to be lowered in order to provide the appropriate voltage levels expected at the input of a consecutive block. The required shift of the single-ended voltages has the effect of adding a different constant to the corresponding log-likelihood values without changing the differential characteristic, i.e., the log-likelihood ratios, see (6.46). Second, the effect of a current drawn by a consecutive block on the voltage outputs of the current block needs to be minimized. A typical effect of such a leakage current in the node processor is a decreased voltage swing at the output of the block which causes a distortion of the output signals.

In the following we discuss three different types of output stages for the example of the Boxplus circuit in Section 6.1.5.

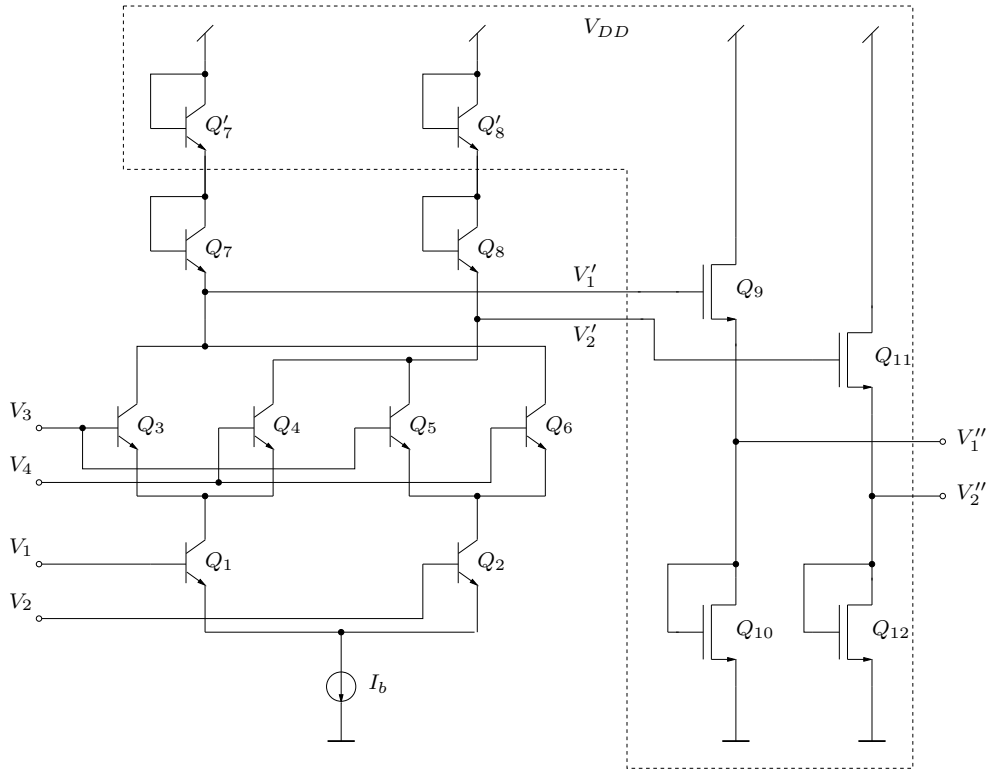



**Figure 6.12**: The (modified) Boxplus circuit with CMOS voltage-dividing output stage.

### 6.3.1 CMOS Voltage-Dividing Output Stage

The first type of output stage is the CMOS voltage-dividing output stage shown in Fig. 6.12. Note that this output stage requires the Boxplus circuit to be expanded with two additional diode-connected transistors $Q'_7$ and $Q'_8$. The reason for this is explained shortly. The left part of the output stage formed of transistors $Q_9$ and $Q_{10}$ can be approximated for saturated devices by

$$\frac{(V''_1 - V_{th_{10}})}{(V'_1 - V''_1 - V_{th_9})} = \sqrt{\frac{\mathrm{W}_9\mathrm{L}_{10}}{\mathrm{L}_9\mathrm{W}_{10}}}, \tag{6.60}$$

which relates the input voltage $V'_1$ and the output voltage $V''_1$ to the device sizes and the corresponding threshold voltages $V_{th_9}$ and $V_{th_{10}}$. A similar expression can be derived for the right part of the output stage consisting of devices $Q_{11}$ and $Q_{12}$. It can be concluded from (6.60) that the single-ended voltage levels of the Boxplus circuit are lowered by means of a voltage division which consequently also reduces the (differential) voltage swing. For equal device sizes of $Q_9$ to $Q_{12}$ and equal threshold voltages it can be found that $V''_1/V'_1 = 1/2$ and $V''_2/V'_2 = 1/2$. This scaling effect can be compensated for by increasing, i.e., doubling, the voltage swing at the input of the buffer stage. This is achieved in Fig. 6.12 by using two diode loads per output instead of only one.

The devices in the output stage may be sized differently in order to minimize the error of the overall building block, e.g., due to the effect of short channel lengths in the output stage. Furthermore, one or more consecutive building blocks may pull load currents from the output stage which then also need to be provided by $Q_9$ and $Q_{11}$. This effect can also be partly compensated for by adjusting the device sizes in the output stage.

This CMOS voltage-dividing output stage has been utilized in our BiCMOS decoder implementation described in Section 7.2.

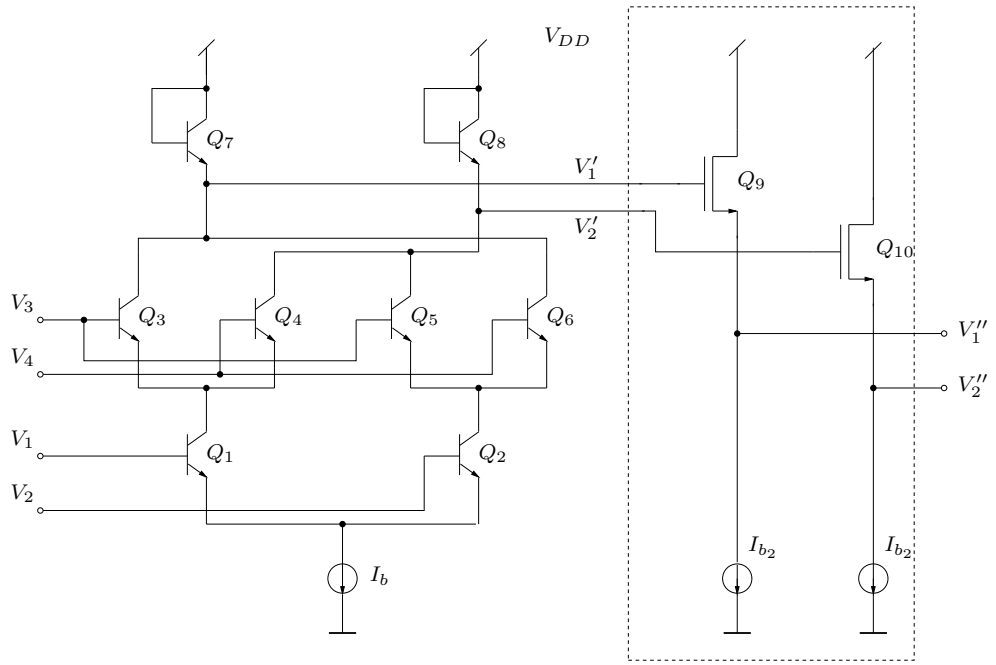



**Figure 6.13**: The Boxplus circuit with CMOS voltage-shifting output stage.

### 6.3.2 CMOS Voltage-Shifting Output Stage

The next output stage is the CMOS voltage-shifting output stage shown in Fig. 6.13. It is very similar to the voltage-dividing output stage in Fig. 6.12. The main difference is that the two diode-connected transistors $Q_{10}$ and $Q_{12}$ in Fig. 6.12 are replaced by two current sources $I_{b_2}$ which dominate the drain currents of $Q_9$ and $Q_{10}$. In this case no modification of the Boxplus circuit is required, i.e., there is only one diode-connected transistor for each output. This is because this output stage generates a voltage shift without affecting the voltage swing. The achievable voltage shift is determined by the gate-source voltage drop of $Q_9$ and $Q_{10}$. The gate-source voltage drop of $Q_9$ is given for saturated devices as

$$V_{GS_9} = \sqrt{\frac{2I_{b_2}\,\mathrm{L_9}}{\mu\,\mathrm{Cox}\mathrm{W_9}}} + V_{th_9}. \tag{6.61}$$

An equivalent expression can be derived for the right part of the output stage. The voltage shift can be adjusted by changing the device sizes in the output stage and the bias current $I_{b_2}$. These parameters can be optimized in the design process in order to minimize the overall error of the building block and to reduce the effect of load currents pulled from the output stage.

### 6.3.3 Bipolar Voltage-Shifting Output Stage

The CMOS voltage-shifting output stage in Fig. 6.13 can also be realized using bipolar transistors as depicted in Fig. 6.14. The voltage shift of this configuration is determined by the base-emitter voltage drop of transistors $Q_9$ and $Q_{10}$, which is in the order of 0.7 to 0.8 V depending on the bias current $I_{b_2}$. The main disadvantage of such a bipolar voltage level shifter is due to the currents drawn by the output stage which are not negligible as it was the case for

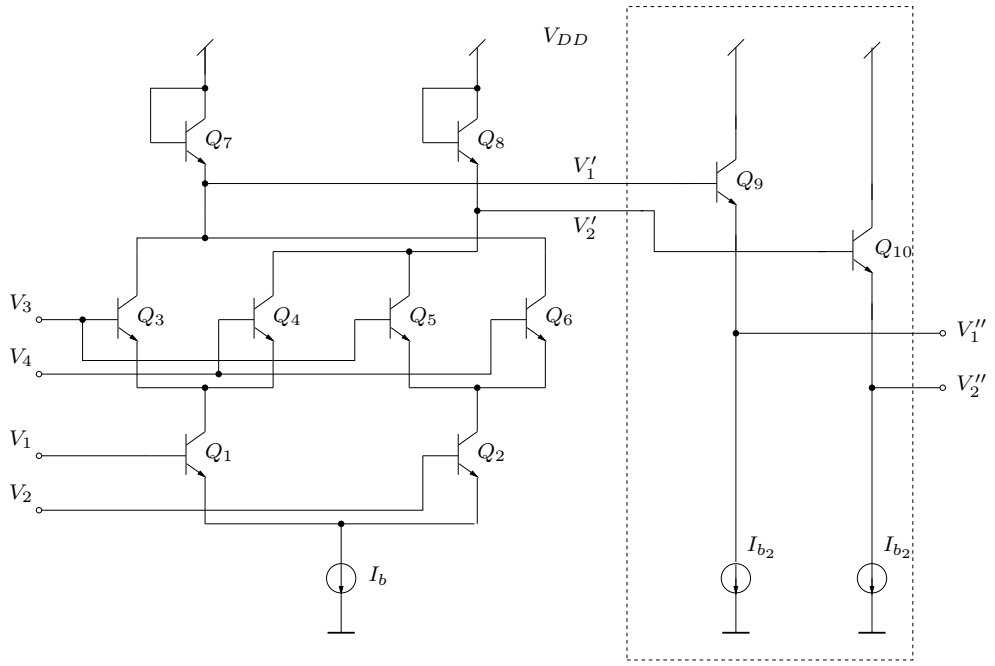



**Figure 6.14**: The Boxplus circuit with bipolar voltage-shifting output stage.

the CMOS output stages discussed in Sections 6.3.1 and 6.3.2. When we neglect a load current pulled from the output stage these currents amount to $I_{b_2}/\beta$ and need to be provided by the diode-connected transistors in the Boxplus circuit. This causes a reduced (differential) voltage swing at the output.

A solution for this problem is to compensate for the base current drawn by the output stage. This can be achieved by means of the additional transistors $Q_{11}$ to $Q_{15}$ as shown in Fig. 6.15. In order to generate the required compensation current we apply a copy of the bias current $I_{b_2}$ used in the output stage to transistor configuration $Q_{11}$ and $Q_{12}$. The collector current of transistor $Q_{12}$ then equals $I_{b_2}/\beta$. This current is applied to the pMOS current mirrors $Q_{13}$, $Q_{14}$ and $Q_{15}$, which copy the current to the input of the buffer stage. The overall error of the building block can be further minimized by a small variation in this compensation current. This is can easily be achieved by selecting a slightly different transistor size for $Q_{13}$ than for $Q_{14}$ and $Q_{15}$.

The bipolar voltage-shifting output stage with load current compensation has been utilized in our SiGe decoder implementation described in Section 7.3.

## 6.4 Decoder Examples

The decoder examples in this section are intended to give some insight into the basic principles in order to allow the reader to develop decoding networks for arbitrary codes. The previous sections covered building blocks and different interfacing circuits for analog decoders. We now demonstrate how these blocks can be used in order to construct some basic analog decoding networks. We in particular highlight the required interfacing and voltage level shifting involved between the building blocks. In principle, any of the voltage level shifters from Section 6.3 can be used. In the following we assume, however, that the output of a voltage level shifter provides the appropriate voltage levels for the upper input of a consecutive building block. We start with check node and variable node decoders which can be employed for decoding SPC codes and repetition codes, respectively. We then turn our focus to a decoder for convolutional codes. These decoder examples can be utilized as component decoders for state-of-the-art turbo codes and LDPC codes.

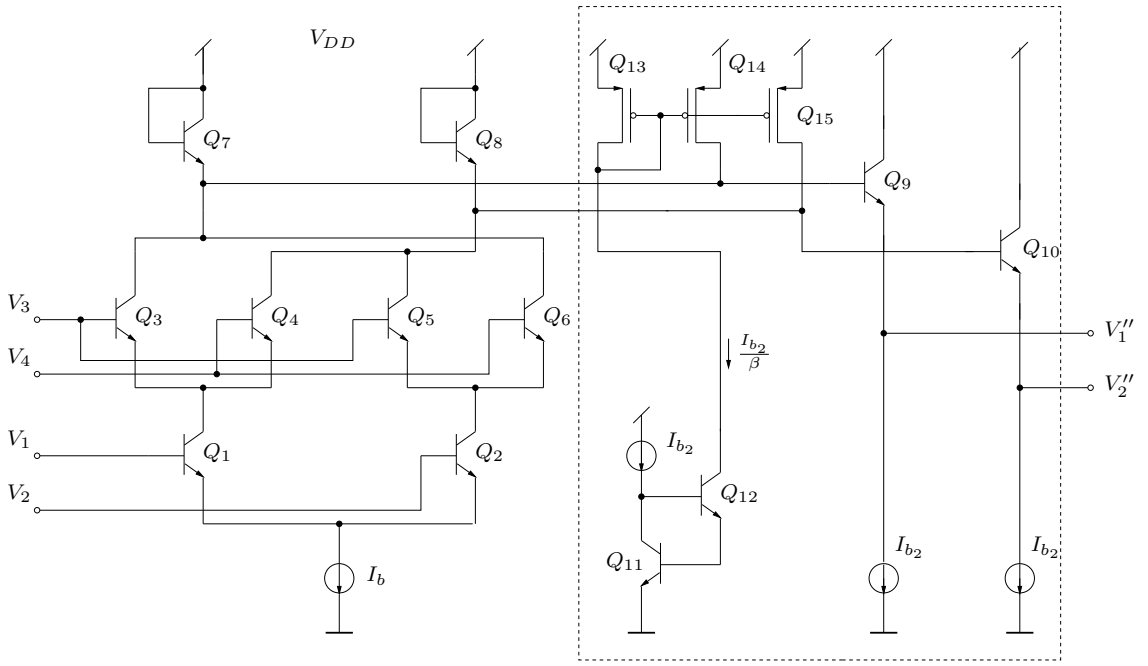



**Figure 6.15**: The Boxplus circuit with bipolar voltage-shifting output stage including load current compensation.

### 6.4.1 Check Node and Variable Node Decoders

Our first example of an analog decoding network is a check node decoder for a $(N, N-1, 2)$ SPC code. The decoder is best described by a normal graph representation of the code, where the degree of the nodes is limited to three as shown in the lower part of Fig. 6.16. One check node processors is exposed as block diagram in the upper part of Fig. 6.16. Note that Fig. 6.16 is essentially identical to the check node decoder in Fig. 5.5. The only difference is that two equality constraint nodes become redundant in the following decoder implementation. The channel value $L_c y$ corresponds to code bit $X$ and represents the input to the check node processor. The $\alpha$ and $\beta$ modules perform the forward and backward recursions within the check node processor while the $\sigma$ module calculates the extrinsic decoder output $L_e(X)$. The decoder internal log-likelihood ratios $L_\alpha(S)$ and $L_\beta(S')$ represent the results of the Boxplus operation of all preceding and succeeding decoder input values, respectively, while $L_\alpha(S') = L_\alpha(S) \boxplus L_c y$ and $L_\beta(S) = L_c y \boxplus L_\beta(S')$.

In the following we focus on the circuit implementation of a check node processor as shown in the upper part of Fig. 6.16. The input $L_c y$ is represented as voltage vector

$$\boldsymbol{V}_{L_c y} = \left[ V_{L_c y}^{(1)} \; V_{L_c y}^{(2)} \right], \text{ with } V_{L_c y}^{(1)} - V_{L_c y}^{(2)} = V_T \, L_c y. \tag{6.62}$$

Note that the channel value is represented by the differential input voltage and not the single-ended input voltages $V_{L_c y}^{(1)}$ and $V_{L_c y}^{(2)}$. These single-ended input voltages can be chosen freely in order to match the input requirements of the module. The voltage vector $\boldsymbol{V}_{L_c y}$ is applied concurrently to the lower inputs (input A) of the $\alpha$ and $\beta$ modules as shown in Fig. 6.17 a). Note that the associated equality node for this input becomes redundant in Fig. 6.16. This is because the lower inputs of the $\alpha$ and $\beta$ modules accept the same vector $\boldsymbol{V}_{L_c y}$ of single-ended input voltages. All replicas at the output of the equality node are then identical to the input and

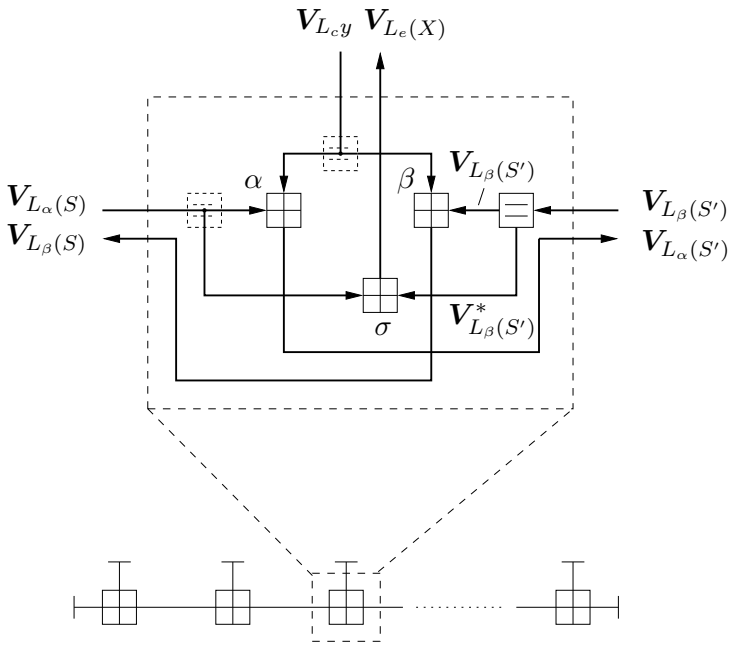



**Figure 6.16**: Block diagram of a check node decoder for a SPC code.

no voltage level shifting is required. The inputs $L_\alpha(S)$ and $L_\beta(S')$ to the $\alpha$ and $\beta$ modules are represented by the voltage vectors

$$\boldsymbol{V}_{L_\alpha(S)} = \left[V^{(1)}_{L_\alpha(S)}\ V^{(2)}_{L_\alpha(S)}\right],\ \text{with}\ V^{(1)}_{L_\alpha(S)} - V^{(2)}_{L_\alpha(S)} = V_T\ L_\alpha(S) \tag{6.63}$$

and

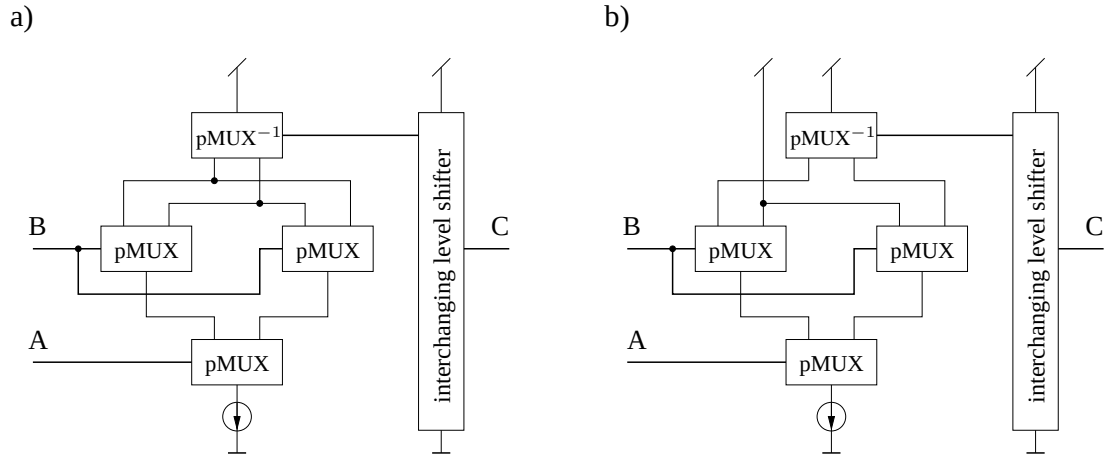
$$\boldsymbol{V}_{L_\beta(S')} = \left[V^{(1)}_{L_\beta(S')}\ V^{(2)}_{L_\beta(S')}\right],\ \text{with}\ V^{(1)}_{L_\beta(S')} - V^{(2)}_{L_\beta(S')} = V_T\ L_\beta(S'). \tag{6.64}$$

These voltage vectors are applied to the upper inputs (input B) of the $\alpha$ and $\beta$ modules as illustrated in Fig. 6.17 a). Furthermore, the inputs in (6.63) and (6.64) also need to be applied to the upper and lower inputs of the $\sigma$ module, respectively. In order to connect to the lower input we need to adjust the single-ended voltage levels of $\boldsymbol{V}_{L_\beta(S')}$. This voltage level shift is achieved through the equality module in Fig. 6.18 which is represented as equality constraint node in Fig. 6.16. Here, the output voltage levels are shifted by the DC voltage drop $\Psi$ across the resistor according to

$$\boldsymbol{V}^*_{L_\beta(S')} = \left[V^{(1)}_{L_\beta(S')} - \Psi\ V^{(2)}_{L_\beta(S')} - \Psi\right], \tag{6.65}$$

without changing the differential characteristic. The computational cores of the $\alpha$, $\beta$ and $\sigma$ modules in Fig. 6.17 a) are identical to the Boxplus circuit introduced in Section 6.1.5. The computation is followed by an output stage as described in Section 6.3. Note that each pair of output wires needs to be interchanged in order maintain the correct polarity of the differential output voltages, see Section 6.2.2 and the definition of the output voltage in (6.42). Such a stage is thus referred to as interchanging level shifter. The $\alpha$ and $\beta$ modules provide the output voltage vectors $\boldsymbol{V}_{L_\alpha(S')}$ and $\boldsymbol{V}_{L_\beta(S)}$, respectively. Each voltage vector consists of two single-ended voltages as in (6.63) and (6.64) which are appropriate for input B of a consecutive module. The output $L_e(X)$ of the check node processor is calculated in the $\sigma$ module and represented as voltage vector

$$\boldsymbol{V}_{L_e(X)} = \left[V^{(1)}_{L_e(X)}\ V^{(2)}_{L_e(X)}\right],\ \text{with}\ V^{(1)}_{L_e(X)} - V^{(2)}_{L_e(X)} = V_T\ L_e(X). \tag{6.66}$$




| | input A | input B | input C |
|---|---|---|---|
| $\alpha$ module | $\boldsymbol{V}_{L_c y}$ | $\boldsymbol{V}_{L_\alpha(S)}$ | $\boldsymbol{V}_{L_\alpha(S')}$ |
| $\beta$ module | $\boldsymbol{V}_{L_c y}$ | $\boldsymbol{V}_{L_\beta(S')}$ | $\boldsymbol{V}_{L_\beta(S)}$ |
| $\sigma$ module | $\boldsymbol{V}^*_{L_\beta(S')}$ | $\boldsymbol{V}_{L_\alpha(S)}$ | $\boldsymbol{V}_{L_e(X)}$ |

**Figure 6.17**: Schematic of the $\alpha$, $\beta$ and $\sigma$ modules in a check node processor in a) and an equality node processor in b).

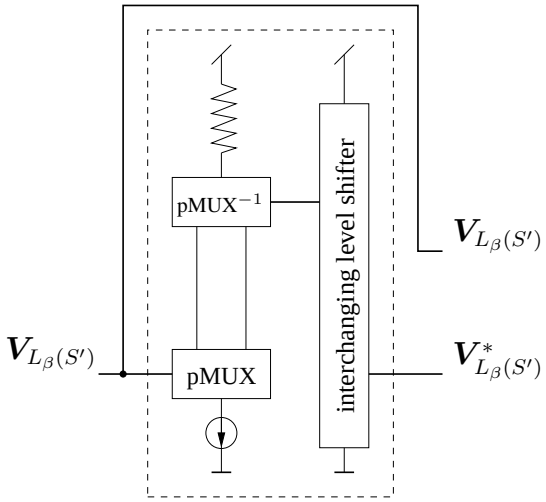



**Figure 6.18**: Schematic of the equality module in the SPC decoder (voltage level shifter).

A general variable node decoder for a $(N, 1, N)$ repetition code can be obtained by simply replacing the check node processor in Fig. 6.17 a) with the corresponding equality node processors as shown in Fig. 6.17 b). This module is identical to the summation circuit introduced in Section 6.1.6.

We now investigate the complexity of a check node processor (CNP) and a variable node processor (VNP). Both bipolar technology and CMOS technology is considered. For the bipolar implementation we employ the voltage-shifting output stage with load current compensation from Fig. 6.15 while the CMOS implementation uses the CMOS equivalent of the voltage-shifting output stage in Fig. 6.14. The transistor count of the individual modules and the overall check node or variable node processor is summarized in Table 6.2. The bipolar implementation requires 56 transistors[1] while 39 devices are sufficient in case of CMOS (transistors for biasing the circuits are included). The 30 percent smaller transistor count in the CMOS implementation is due to less transistors in the output stage (-2) and the omitted equality module (-11).

**Table 6.2**: Transistor count of a check node processor (CNP) and a variable node processor (VNP).

| | # of transistors | |
|---|---|---|
| | bipolar technology | CMOS technology |
| $\alpha$ module | 15 | 13 |
| $\beta$ module | 15 | 13 |
| $\sigma$ module | 15 | 13 |
| equality module | 11 | - |
| **CNP / VNP** | **56** | **39** |

### 6.4.2 Convolutional Decoder

The second decoder example is for a rate $R = 1/2$ convolutional code with memory two and generator polynomials (7,5). A tailbiting version of this code was already introduced in Section 5.2.3. We restrict ourselves to a single node processor for trellis section $k$ as shown in Fig. 6.19. The overall convolutional decoder is then simply obtained by connecting as many such node processors together as there are nodes in the code graph. When the beginning and the end of the code graph is connected together as shown in Fig. 5.13 we obtain a tailbiting convolutional decoder. With an appropriate initialization of the forward and backward recursions at the beginning and the end of the decoder we obtain a decoder for a terminated convolutional code, or, in case only a small fragment of the overall code graph is implemented, a SwinDec decoder. Two possible implementations of a node processor are depicted in Fig. 6.19. Note that the two block diagrams are essentially identical to Fig. 5.14. The only difference is that most of the equality constraint nodes in Fig. 5.14 become redundant in the following decoder implementation.

We first start with a description of the node processor in Fig. 6.19 a) which can be used for convolutional codes with systematic feedback encoders. The channel values $L_c y_k^{(1)}$ and $L_c y_k^{(2)}$ represent the input to the node processor. The $\gamma$ module performs the branch metric calculations and the $\alpha$ and $\beta$ modules perform the forward and backward recursions within the node processor. The $\sigma_1$ and $\sigma_2$ modules are used in order to calculate the decoder output. We assume that the decoder is used as component decoder in a turbo scheme so that only the extrinsic information $L_e(U)$ needs to be calculated.

[1] Plus one resistor. No resistors are required in the CMOS implementation.

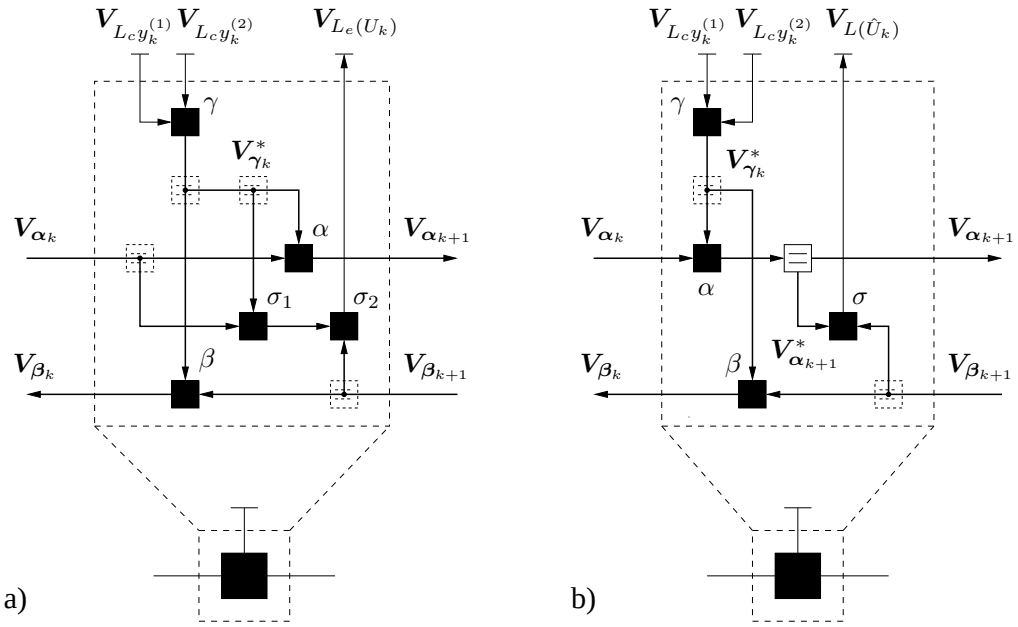



**Figure 6.19**: Block diagram of a node processor for convolutional codes with systematic feedback encoders in a) and feedforward encoders in b).

The inputs $L_c y_k^{(1)}$ and $L_c y_k^{(2)}$ are represented by the voltage vectors

$$\boldsymbol{V}_{L_c y_k^{(i)}} = \left[ V^{(1)}_{L_c y_k^{(i)}} \; V^{(2)}_{L_c y_k^{(i)}} \right], \text{ with } V^{(1)}_{L_c y_k^{(i)}} - V^{(2)}_{L_c y_k^{(i)}} = V_T \, L_c y_k^{(i)}, \tag{6.67}$$

with $i \in \{1, 2\}$. Note that the L-values obtained from the channel only determine the differential characteristic of the input voltages and not the single-ended input voltages. The single-ended input voltages can be chosen freely in order to match the input requirements of the module. The two voltage vectors in (6.67) are applied to the $\gamma$ module in Fig. 6.19. The circuit implementation of the $\gamma$ module is depicted in Fig. 6.20. It is assumed that the voltage levels are appropriate in order to connect to the lower and upper inputs of the module. The $\gamma$ module calculates the four branch metrics associated with the trellis section according to (4.32) and (4.31). The computation of the branch metrics is followed by a voltage level shifter as introduced in Section 6.3 with four inputs and outputs. Note that the level shifter maintains the same order of input and output wires. It is followed by an inverter stage consisting of pMUX and pMUX$^{-1}$ in order to obtain the desired output voltages. This inverter stage becomes necessary because the single-ended output voltages of the branch metric computation are proportional to the negative logarithm of the probabilities so that they cannot directly be applied to the input of another module. Note that in the binary case of the SPC decoder in Section 6.4.1 it was sufficient to simply interchange the output wires. A resistor is added on top of the pMUX$^{-1}$ in the inverter stage in order to lower the single-ended voltage levels at the output by the DC voltage drop across the resistor, see Fig. 6.18 and (6.65). This is because the output of the $\gamma$ module $\boldsymbol{V}^*_{\gamma_k}$ is connected to the lower inputs of the $\alpha$, $\beta$ and $\sigma_1$ modules.

The $\alpha$ and $\beta$ modules in Fig. 6.19 perform the calculations in (4.40) and (4.41), respectively. The circuit implementation of these modules is depicted in Fig. 6.21 and Fig. 6.22. The upper inputs of these modules are determined by the voltage vectors $\boldsymbol{V}_{\boldsymbol{\alpha}_k}$ and $\boldsymbol{V}_{\boldsymbol{\beta}_{k+1}}$ which represent the four state metrics in the forward and backward recursions of the decoder. More precisely, $\boldsymbol{V}_{\boldsymbol{\alpha}_k}$ and $\boldsymbol{V}_{\boldsymbol{\beta}_{k+1}}$ represent vectors $\boldsymbol{\alpha}_k$ and $\boldsymbol{\beta}_{k+1}$ from (4.38) and (4.39) in the log-likelihood domain including a common DC voltage offset as in (6.50). The main difference between the $\alpha$ and $\beta$ modules lies in the wiring network on top of the analog multiplier. Whenever output

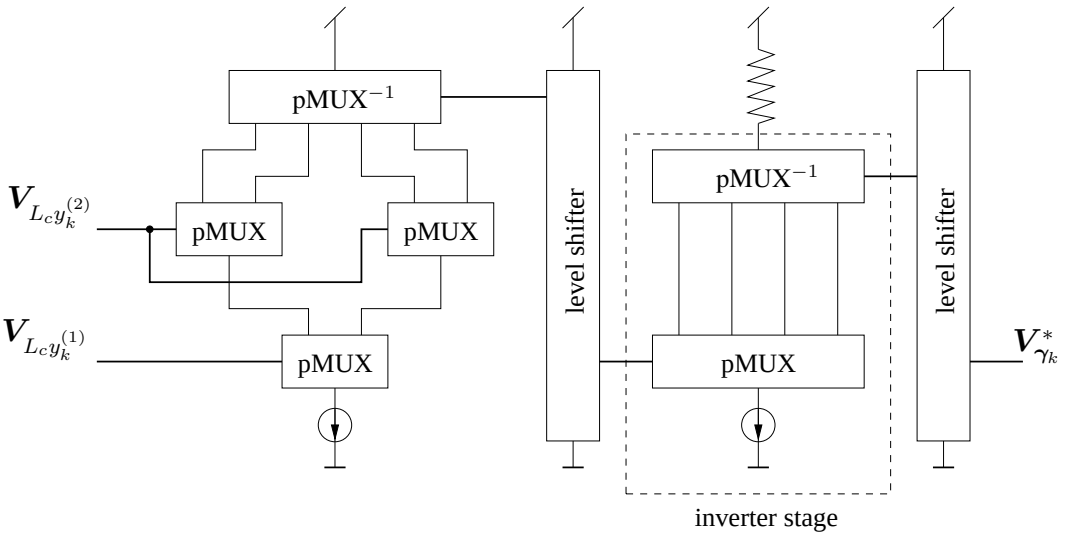



**Figure 6.20**: Schematic of the $\gamma$ module in the convolutional decoder.

currents (calculated probability products) do not contribute to the output of the module, i.e., the corresponding transitions are not present in the trellis diagram, the output is connected to the supply voltage. Similar to the $\gamma$ module, an inverter stage is used in both the $\alpha$ and $\beta$ modules. The voltage outputs $\boldsymbol{V}_{\boldsymbol{\alpha}_{k+1}}$ and $\boldsymbol{V}_{\boldsymbol{\beta}_k}$ of these modules are used as upper inputs for neighboring node processors so that there is no resistor on top of the inverter stage. Note that no probability normalization is required at the output of the $\alpha$ and $\beta$ modules. This is due to the use of differential output voltages where any scaling cancels out.

The extrinsic output of the decoder is calculated in the $\sigma_1$ and $\sigma_2$ modules. The calculation is based on the inputs from the forward and the backward recursions of neighboring node processors and the output of the $\gamma$ module. The circuit implementation of these modules is depicted in Fig. 6.23 a) and Fig. 6.23 b). The $\sigma_1$ module calculates eight probability products based on the voltage vector $\boldsymbol{V}_{\boldsymbol{\alpha}_k}$ from the forward recursion and the voltage vector $\boldsymbol{V}^*_{\boldsymbol{\gamma}_k}$ provided by the $\gamma$ module. Note that $\boldsymbol{V}^*_{\boldsymbol{\gamma}_k}$ generates four output currents in the corresponding pMUX which need to be summed up pairwise as shown in Fig. 6.23 a) in order to yield the desired behavior. This is because $L_c y_k^{(1)}$ does not contribute to the calculation of the extrinsic information.[2] Similar to the $\gamma$ module a resistor is added on top of the inverter stage in order to lower the output voltage levels so that the output can be connected to the lower input of the $\sigma_2$ module in Fig. 6.23 b). The $\sigma_2$ module multiplies each of the eight probabilities with the corresponding $\beta$ values thus generating a total of 32 probability products while only eight of them contribute to the decoder output. The voltage vectors $\boldsymbol{V}_{\boldsymbol{\alpha}_k}$, $\boldsymbol{V}_{\boldsymbol{\beta}_{k+1}}$ propagate through the convolutional decoder and the output voltages in $\boldsymbol{V}_{L_e(U_k)}$ attain steady-state values after the settling time of the network. The extrinsic output $L_e(U_k)$ of the convolutional decoder is then represented by the voltage vector

$$\boldsymbol{V}_{L_e(U_k)} = \left[ V^{(1)}_{L_e(U_k)} \; V^{(2)}_{L_e(U_k)} \right], \text{ with } V^{(2)}_{L_e(U_k)} - V^{(1)}_{L_e(U_k)} = V_T \, L_e(U_k). \tag{6.68}$$

Note that the sign inversion in the calculation of the differential output voltage in (6.68) which is due to the omitted interchanging level shifter in Fig. 6.23 b). The overall decoder output can be obtained by simply adding the L-value of the corresponding systematic code bit, i.e., $L(\hat{U}_k) = L_e(U_k) + L_c y_k^{(1)}$. Alternatively, $L(\hat{U}_k)$ could be calculated directly with slightly modified $\sigma_1$ and $\sigma_2$ modules.

[2] Alternatively, the channel information about the second code bit $\boldsymbol{V}_{L_c y_k^{(2)}}$ could be used as lower input to the $\sigma_1$ module (after appropriate voltage level-shifting).

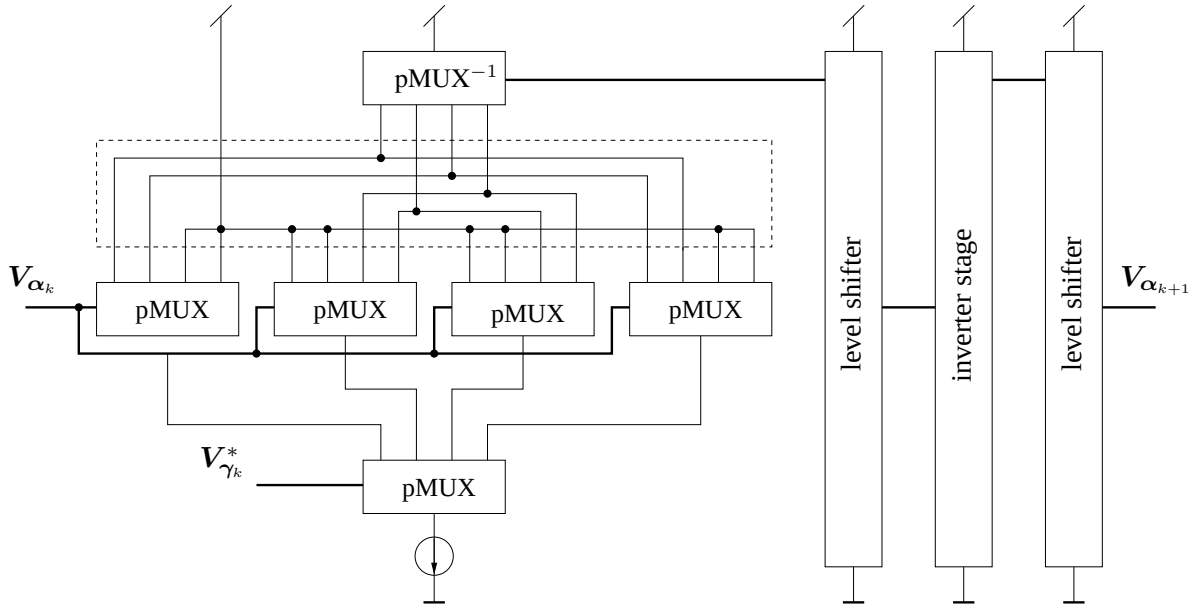



**Figure 6.21**: Schematic of the $\alpha$ module in the convolutional decoder.

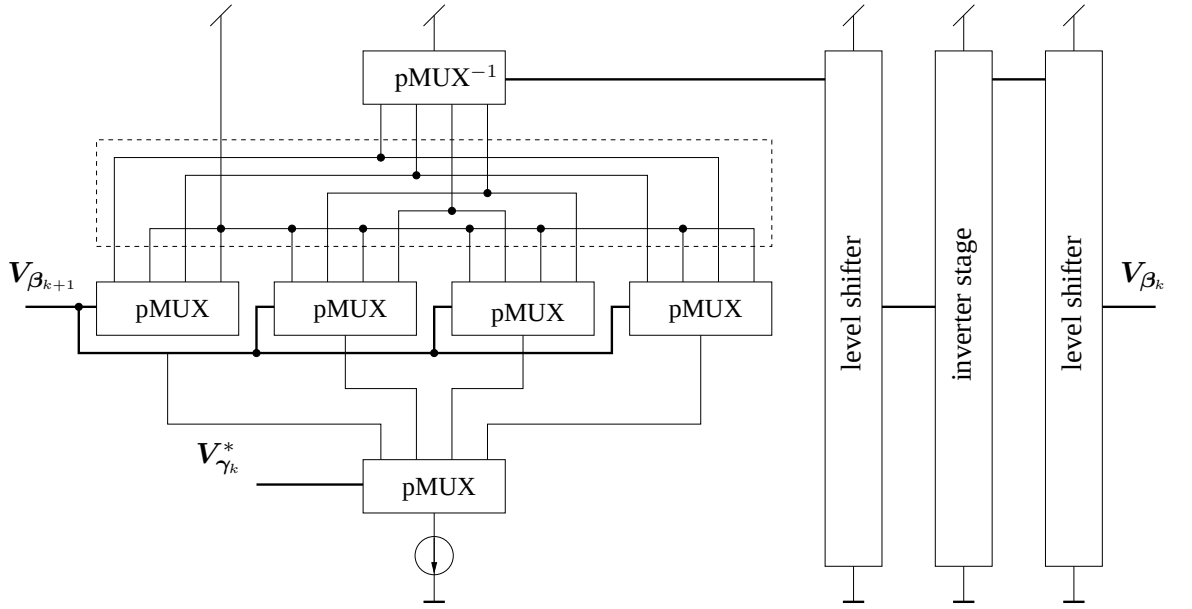



**Figure 6.22**: Schematic of the $\beta$ module in the convolutional decoder.

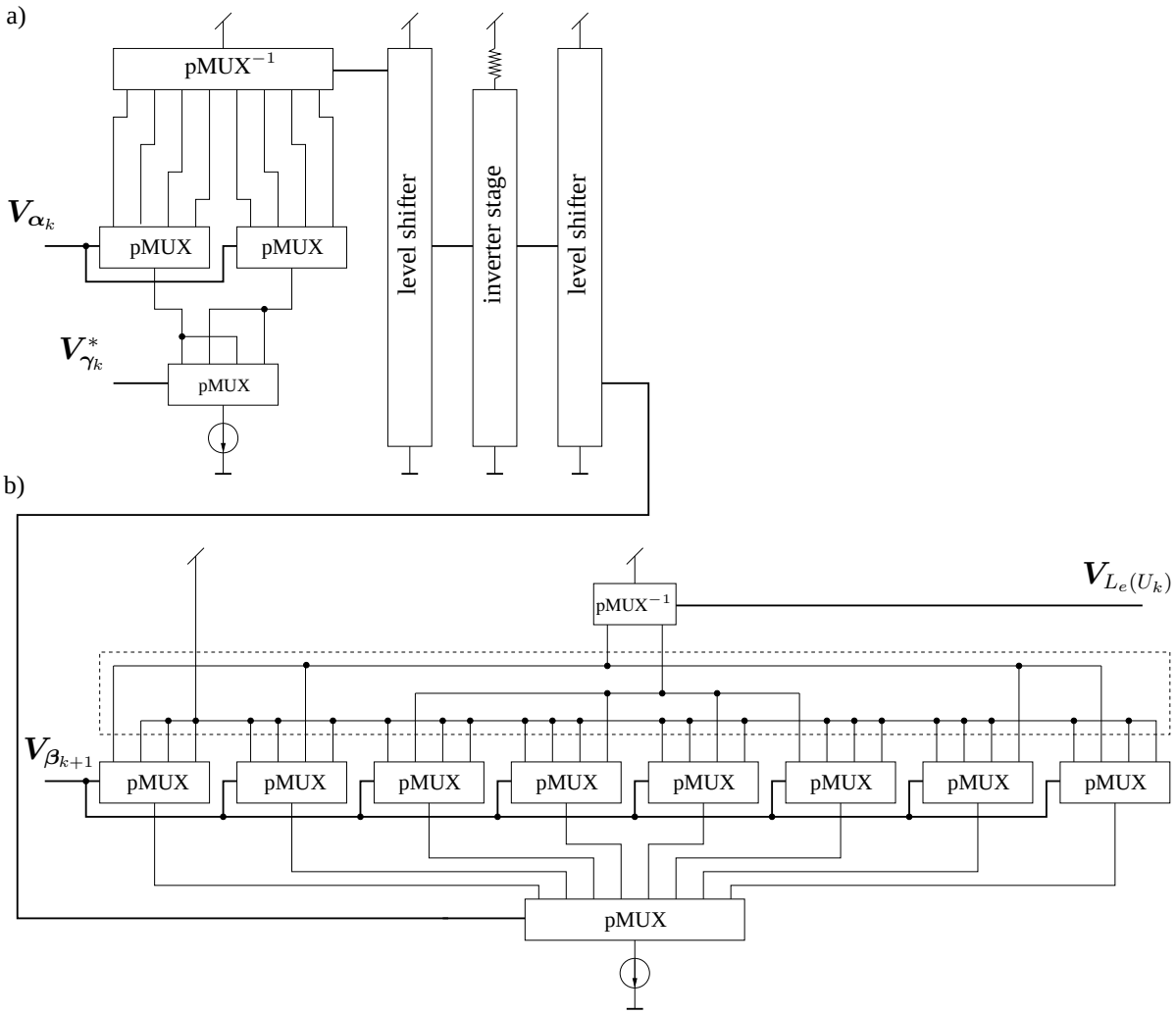



**Figure 6.23**: Schematic of the $\sigma_1$ module in a) and the $\sigma_2$ module in b) for the calculation of the extrinsic output in a convolutional decoder (systematic feedback encoder).

The implementation of the node processor in Fig. 6.19 b) can be facilitated for convolutional codes with feedforward encoders. This node processor uses identical $\gamma$, $\alpha$ and $\beta$ modules as the node processor in Fig. 6.19 a) and calculates the overall decoder output $L(\hat{U}_k)$ instead of the extrinsic output. It exploits the fact that for feedforward encoders the information bit appears in state $S'$ of trellis section $k$. This allows the calculation of the decoder output in the $\sigma$ module based on the voltage vectors $\boldsymbol{V}^*_{\boldsymbol{\alpha}_{k+1}}$ and $\boldsymbol{V}_{\boldsymbol{\beta}_{k+1}}$. The implementation of this $\sigma$ module is depicted in Fig. 6.24. For this, the voltage output $\boldsymbol{V}_{\boldsymbol{\alpha}_{k+1}}$ of the $\alpha$ module needs to be shifted to lower voltage levels. This is simply achieved with an additional output stage in the equality node of Fig. 6.19 b) rather than the equality module in Fig. 6.18. Note that the $\sigma$ module of this node processor has a significantly lower complexity than the $\sigma_1$ and $\sigma_2$ modules in other node processor which consequently decreases overall complexity.

The complexity of the two node processors in Fig. 6.19 is now further investigated for the more general case of rate $R = 1/2$ codes with memory $m$. This allows us to easily scale the complexity to other code memories. We assume a bipolar technology and the use of voltage-shifting output stages with load current compensation from Fig. 6.15. The transistor count of the individual modules and the overall node processors is summarized in Table 6.3. It is apparent that the complexity of the node processor increases exponentially with the memory of

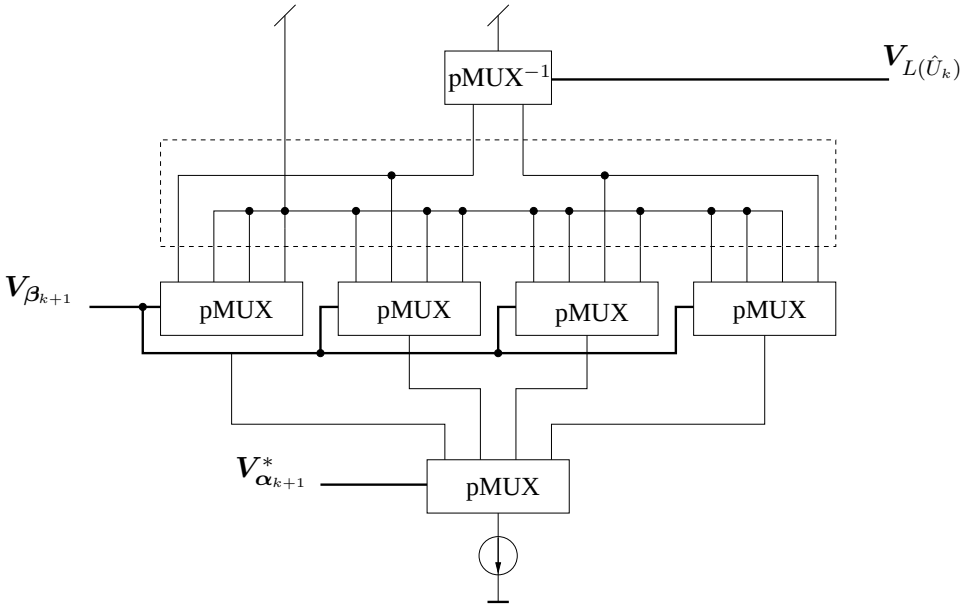



**Figure 6.24**: Schematic of the $\sigma$ module for the calculation of the decoder output in a convolutional decoder (feedforward encoder).

**Table 6.3**: Transistor count of different node processors for rate $R = 1/2$ convolutional codes with memory $m$.

| module | core | shifter 1 | inverter | shifter 2 | total |
|---|---|---|---|---|---|
| $\gamma$ | 11 | 12 | 9 | 12 | 44 |
| $\alpha$ | $5\,(2^m+1)$ | $3\cdot 2^m$ | $2\cdot 2^m+1$ | $3\cdot 2^m$ | $13\cdot 2^m+6$ |
| $\beta$ | $5\,(2^m+1)$ | $3\cdot 2^m$ | $2\cdot 2^m+1$ | $3\cdot 2^m$ | $13\cdot 2^m+6$ |
| $\sigma_1$ | $4\cdot 2^m+5$ | $6\cdot 2^m$ | $4\cdot 2^m+1$ | $6\cdot 2^m$ | $20\cdot 2^m+6$ |
| $\sigma_2$ | $2\left(2^{2m}+2^m\right)+3$ | - | - | - | $2\left(2^{2m}+2^m\right)+3$ |
| $\sigma$ | $2^{2m}+2^m+3$ | - | - | - | $2^{2m}+2^m+3$ |
| equality | - | $3\cdot 2^m$ | - | - | $3\cdot 2^m$ |
| **node processor in Fig. 6.19 a)** with $\gamma$, $\alpha$, $\beta$, $\sigma_1$ and $\sigma_2$ modules | | | | | $\mathbf{2\cdot 2^{2m}+48\cdot 2^m+65}$ |
| **node processor in Fig. 6.19 b)** with $\gamma$, $\alpha$, $\beta$, equality and $\sigma$ modules | | | | | $\mathbf{2^{2m}+30\cdot 2^m+59}$ |

**Table 6.4**: Evaluation of the transistor count of different node processors for rate $R = 1/2$ convolutional codes with memory $m$.

| code memory | transistor count | |
|---|---|---|
| | node processor in Fig. 6.19 a) | node processor in Fig. 6.19 b) |
| $m=2$ | 289 | 195 |
| $m=3$ | 577 | 363 |
| $m=4$ | 1345 | 795 |
| $m=5$ | 3649 | 2043 |
| $m=6$ | 11329 | 6075 |

the code. With the memory $m = 2$ code from above we obtain 289 transistors[3] for the node processor in Fig. 6.19 a) and 195 transistors[4] for the node processor in Fig. 6.19 b) (transistors for the biasing of the circuits are included). Further code memories are evaluated in Table 6.4. Note that a CMOS implementation typically requires a smaller number of transistors, see Section 6.4.1.

### 6.4.3 Complexity of Turbo Decoders and LDPC Decoders

In the following we analyze the complexity of the different decoder architectures introduced in Section 5.4. We restrict ourselves to the complexity of the analog decoder core in terms of the transistor count. This number is particularly interesting since it allows the estimation of the overall chip area for a given technology. This estimation is particularly suited for fully parallel decoder architectures where the overall chip area is dominated by the transistor count in the decoder core. In case of a sliding window turbo decoder we need to take additional control and storage elements into account and, depending on the realization of these storage elements, may also need to consider internal A/D and D/A converters. The decoder examples in Sections 6.4.1 and 6.4.2 are chosen so that they can directly be facilitated as component decoders. The complexity analysis relies on the transistor count of individual check node and variable node decoders in Table 6.2 and the transistor count of a node processor in a convolutional decoder as summarized in Table 6.3.

We start with a fully parallel turbo decoder for the popular example of the rate $R = 1/3$ UMTS turbo code [ETS00] which is based on the parallel concatenation of two rate $R = 1/2$ convolutional codes with systematic feedback encoder. This example is particularly interesting since this turbo code is not only standardized for UMTS, but also commonly used as reference for the evaluation of turbo codes or the comparison between turbo codes and LDPC codes. The overall bipolar transistor count of a node processor for one trellis section is specified in Table 6.3 with $2 \cdot 2^{2m} + 48 \cdot 2^m + 65$. A node processor for the required memory $m = 3$ convolutional decoder then consists of 577 transistors. The supported block lengths for the turbo code in UMTS range between 40 and 5114 information bits. The number of node processors in each component decoder can be reasonably well approximated by the number of information bits. This is because the computations at the beginning and the end of the terminated code trellis are less complex. We therefore simply omit the node processors for the termination bits. A fully parallel analog UMTS turbo decoder would hence require between 46.2 k transistors and about 5.9 million transistors for the maximum block length of 5114 information bits. A CMOS implementation of a fully parallel turbo decoder for the 40 bit UMTS turbo code can be found in [VGN+05]. The reported transistor count of the decoder core is 30 k. This number is 35 percent lower than our bipolar transistor count given above. This complexity reduction is comparable to the 30 percent lower transistor count achieved through the CMOS implementation of the check node and variable node decoders in Section 6.4.1. However, both bipolar and CMOS implementations reach its limitations when the block length increases beyond a few hundred information bits.

One of the main disadvantages of the fully parallel turbo decoder is that complexity increases linearly with the block length. With increasing block length of the turbo code the transistor count (and thus chip area) becomes unacceptable. Furthermore, the rather low speed requirements of only 2 Mbps (or 10 Mbps in the HSDPA extension) do by no means necessitate full parallelization in the decoder. Another disadvantage is that every block length requires a dedicated decoder chip with a fixed wiring network for the interleaver which rules out most practical applications. Our solution to these problems is the sliding window turbo decoder ar-

[3] Plus two resistors.
[4] Plus one resistor.

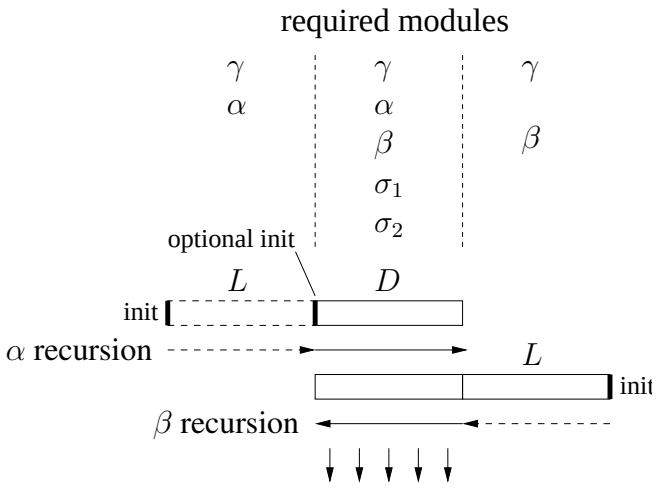



**Figure 6.25**: Required modules for the different sections of the SwinDec decoder.

chitecture presented in Section 5.4.2. We demonstrated that SwinDec component decoders with parameters $D = 8$ and $L = 24$ closely approach the BER performance of the (digital) reference decoder, see Fig. 5.31. The following complexity analysis is based on the SwinDec decoder with an offset of the forward and backward ring as introduced in Section 5.3.3. Due to this offset of the decoding windows not all modules of a node processor contribute in each section of the SwinDec decoder, see Fig. 6.25. Based on the complexity of the individual modules in Table 6.3 there are a total number of

$$2D \cdot 2^{2m} + (48D + 26L)2^m + 65D + 100L \tag{6.69}$$

bipolar transistors in each SwinDec decoder. With $D = 8$, $L = 24$ and $m = 3$ we obtain a total number of about 12 k transistors. Here, we assume a stabilization length for both the forward ($\alpha$) and backward ($\beta$) recursions as shown in Fig. 5.25. The complexity of the SwinDec decoder can be reduced when we store the result of the forward recursion in order to appropriately initialize the forward recursion in the following decoding window. We can then remove the stabilization length in the forward window as indicated with the dashed box in Fig. 6.25. In this case the SwinDec decoder requires

$$2D \cdot 2^{2m} + (48D + 13L)2^m + 65D + 50L \tag{6.70}$$

bipolar transistors. The parameters from above yield a total number of 8310 transistors in each SwinDec decoder. When two of these SwinDec decoders are instantiated in the turbo decoder we then obtain roughly 24 k or 16.6 k transistors depending on the initialization of the $\alpha$ recursion. Hence, the computational complexity is reduced by a factor of 245 or even 355 compared to a fully parallel decoder implementation for the maximum block length. Clearly, the advantages of this architecture come at the expense of additional control hardware and storage elements for the extrinsic information. This memory can be implemented either in the analog or the digital domain. The latter necessitates A/D and D/A converters which are utilized in each iteration of the decoder. Both increase chip area and power consumption. However, the results of our more detailed analysis of A/D and D/A converters in Chapter 8 indicate that the additional area and power consumption of these converters may be rather low in comparison with the analog decoder core. The use of storage elements in the turbo decoder has also the big advantage that routing between the component decoders is eased. This is particularly important since the routing of a large bidirectional interleaver network poses a major challenge in the design process of a fully parallel turbo decoder. Furthermore, different interleaver structures can then simply be

realized by changing the way the memory elements are addressed. Together with the SwinDec decoder we achieve the required flexibility in order to process various different block lengths and thus interleavers with one single decoder chip. Also, the speed of the component decoders and thus the speed of the overall turbo decoder can easily be scaled through parameter $D$ in the SwinDec decoder. This allows us to match the given speed requirements of the application while operating in a region with a good trade-off between power and speed. The sliding window turbo decoder architecture is thus key for addressing commercial applications like UMTS.

An alternative approach of a fully programable interleaver without memory was successfully demonstrated for a turbo code with 16 information bits in [GGG02], [GG03b], [GG03a]. However, this approach does not appear to be feasible for such large block lengths as in UMTS. There is also related work on the complexity reduction of analog turbo decoders which exploits the special structure of some turbo codes [ALS+05], [ALSJ05], [ASLJ06] and [Arz06].

We now turn our focus to the complexity analysis of analog LDPC decoders as introduced in Section 5.4.3. The transistor count of LDPC decoders is dictated by the $(N-K) \times N$ parity-check matrix $\boldsymbol{H}$ on which the decoder is based. We first determine the number of variable node and check node processors in such a decoder. Due to the degree restrictions of analog node processors every column $i$ in $\boldsymbol{H}$ with weight $d_{v,i}$ is split up into $d_{v,i} - 1$ variable node processors. Similarly, every row $j$ in $\boldsymbol{H}$ with weight $d_{c,j}$ is split up into $d_{c,j} - 2$ check node processors. See Fig. 5.34 for the associated normal graph of such a decoding network. We then obtain for the total number of check node processors (CNP) in the decoder

$$\#\text{CNP} = \sum_{j=1}^{N-K} (d_{c,j} - 2)\,. \tag{6.71}$$

Here, $\sum_{j=1}^{N-K} d_{c,j}$ represents the number of ones in the parity-check matrix. Using the abbreviation $c_H = \sum_{j=1}^{N-K} d_{c,j}$ we obtain

$$\#\text{CNP} = c_H - 2(N-K). \tag{6.72}$$

Similarly, the total number of variable node processors (VNP) is determined by

$$\#\text{VNP} = \sum_{i=1}^{N} (d_{v,i} - 1) = c_H - N, \tag{6.73}$$

which allows us to calculate either the extrinsic information $L_e(X)$ or the overall decoder output $L(\hat{X})$ for all code bits as described in Section 5.2.1. Again, $c_H = \sum_{i=1}^{N} d_{v,i}$ represents the number of ones in the parity-check matrix. The overall transistor count of a LDPC decoder for a given code can then easily be determined based on the calculated number of variable node and check node processors in (6.72) and (6.73), respectively, and the number of transistors for each node processor as given in Table 6.2.

Analog LDPC decoders are further investigated for an application in the next generation wireless LAN [IEE05] in Chapter 8.

# 7

# *Manufactured Decoder Chips*

It is common practice in both academia and industry to verify analog integrated circuits through chip implementations and measurement results. This is in contrast to digital circuit design where computer simulations, in general, allow us to accurately predict the performance. Analog implementations typically require several fabrication runs in order to modify and tune the circuits so that optimal performance and robustness is achieved for a given technology. In fact, it can not be taken for granted that the first chip implementation is fully operational. There are several examples available in the literature where the malfunction of a small component or a tiny error in the physical layout of the chip caused an analog decoder to fail. In this chapter we present two prototypes of analog decoders which originated from a joint cooperation with Bell Laboratories, Lucent Technologies. Both decoder chips are for a simple tailbiting convolutional code in order to prove the basic concept of analog decoding. The first decoder was designed for a 0.25 $\mu$m BiCMOS technology as part of a diploma thesis [Moe99] and was then tested as part of this work. The details of this chip implementation including measurement results are given in Section 7.2. The results are also published in [MGYH00]. To the best of our knowledge, this implementation represents the world's first fully operational analog decoder chip. After this successful verification of our circuit design and the basic concepts of analog decoding we designed a second analog decoder chip. The goal of this implementation was to further enhance the performance of the decoder and in particular to boost decoder speed. This goal was achieved by means of an optimized circuit design and also by using a 0.25 $\mu$m SiGe technology which facilitates faster transistors. The details of this SiGe decoder implementation are presented in Section 7.3. This decoder chip appears to be the world's fastest analog decoder chip to date, despite the fact that only eight bits are decoded in parallel, i.e., the level of parallelization is rather low. The core building blocks of both decoder implementations solely consist of bipolar npn transistors because of superior speed and matching characteristics compared to CMOS devices. The CMOS output stages in the building blocks of the BiCMOS implementation have been replaced in the SiGe implementation by npn output stages. All internal signals are fully differential so that temperature and noise effects are minimized. One of the main differences between our circuit design and other work in this area is that we use (differential) voltage signals at the input and the output of the decoder as well as between the individual building blocks. This approach was later also adopted in [ALSJ05], [ALS+05], [ASLJ06] and [Arz06]. Further chip implementations can be found in [LHL+99a], [Lus00], [WDK+01], [WDL+01], [XVG+02], [GG03b], [GG03a], [Gau03], [WDY+04], [NWGS04], [ABM+04], [FLL+04], [Win04], [Ama04], [VGN+05], [ABM+05], [FLMS05], [WNGS06] and [HBP06].

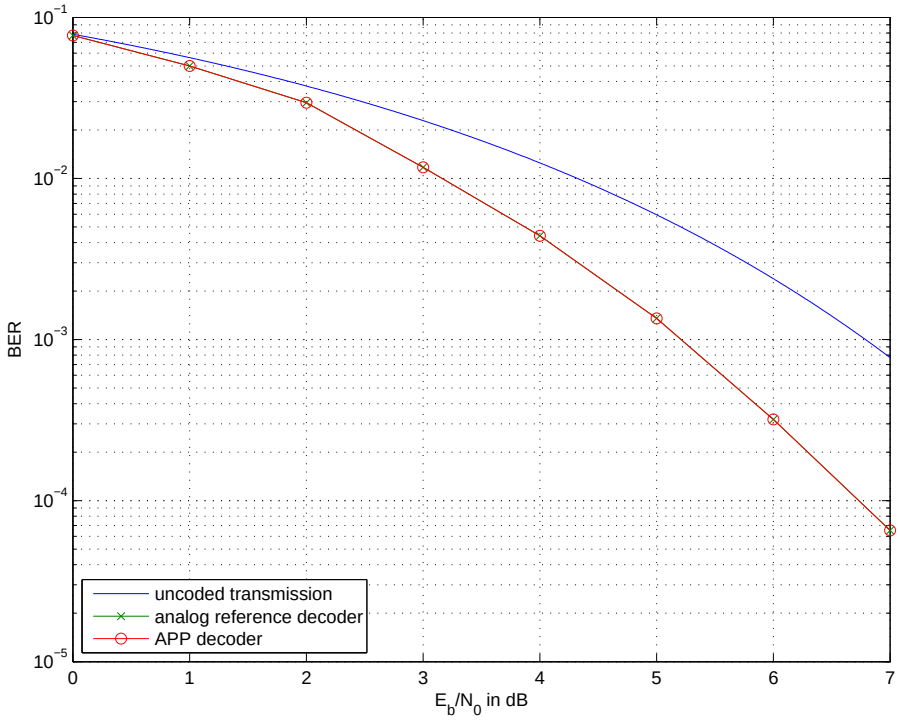



**Figure 7.1**: Simulated BER for the (16,8,3) tailbiting convolutional code.

We start with the introduction of an ideal analog decoder for the considered tailbiting convolutional code. This decoder acts as our reference throughout this chapter. We highlight some important input configurations which were used for the performance evaluation of the decoder chips through circuit-level simulations as well as measurement results. We present the measured BER of both decoder implementations and estimate the speed of the chips based on the measured transient behavior of the chips. The key parameters of our chip implementations are summarized in Section 7.4.

## 7.1 Tailbiting Convolutional Decoder

Both implemented analog decoder chips are for the (16,8,3) tailbiting convolutional code with memory one and two trellis states. A systematic feedforward encoder is assumed. This code has been selected because of its simplicity and regularity in order to verify the basic concept of analog decoding rather than implementing a very powerful decoder with a practical application. The tailbiting structure of the code allows us to exploit fully parallel signal processing in the analog decoder and to investigate the effect of feedback in the analog ring network. This code is also particularly interesting since each trellis section can be implemented as a serial concatenation of a check node and a variable node processor. These node processors are also utilized in analog LDPC decoders and are therefore of particular importance. Fig. 7.1 shows the simulated BER performance of an ideal analog tailbiting convolutional decoder which is used as reference for the evaluation of our chip implementations. This ideal analog decoder is simulated with the time-continuous simulation model in SIMULINK and achieves the BER performance of the corresponding APP/MAP decoder.

In the following two sections we introduce some selected input configurations which will be used for the evaluation of the transient response of the decoder chips.

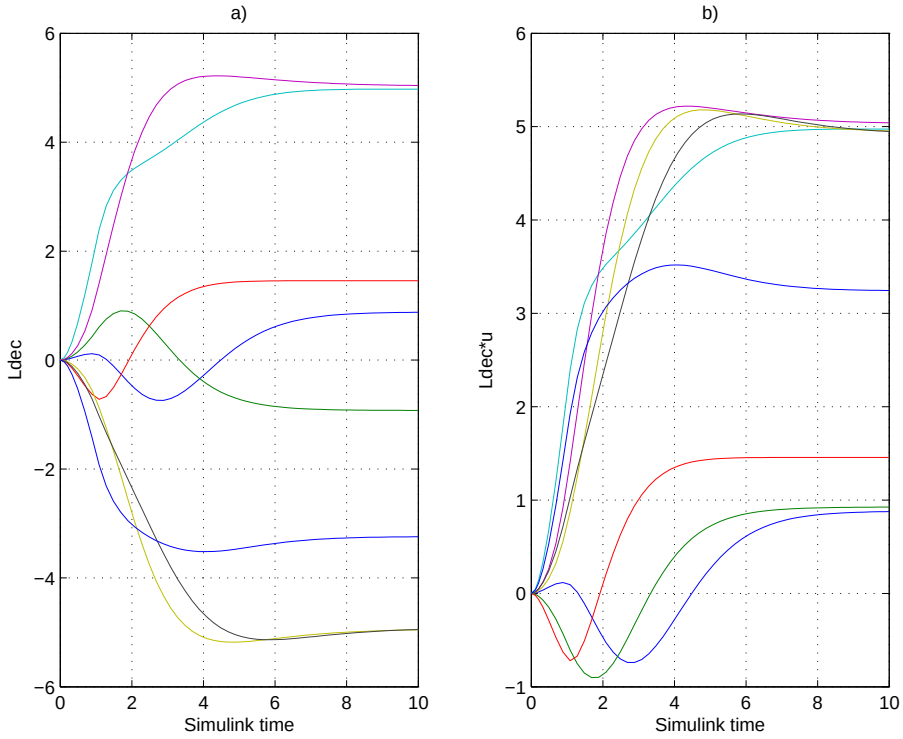



**Figure 7.2**: Transient behavior of the L-values at the output of the ideal analog decoder for the reference input configuration in a) and the L-values weighted with the information word $\boldsymbol{u}$ in b).

### 7.1.1 Reference Input Configuration

The transient behavior of the analog tailbiting convolutional decoders is investigated using an exceptionally input configuration of channel values. This input configuration is particularly demanding since the decoder requires around 5-10 times the average decoding time to correct the last transmission error. We selected such a configuration of channel values out of thousands of high-level simulation runs using random code words. This decoder input is then assumed to lead to the worst-case decoding delay which may be used for speed estimations of the analog decoder chips. This reference input configuration is described by the information word $\boldsymbol{u} = (+1, -1, +1, +1, +1, -1, -1, -1)$ which is mapped onto the code word

$$\boldsymbol{c} = ((+1, -1), (-1, -1), (+1, -1), (+1, +1), (+1, +1), (-1, -1), (-1, +1), (-1, +1)). \tag{7.1}$$

Note that the first bit in each group of code bits equals the corresponding information bit. The associated channel values after transmission over the AWGN channel at a $E_b/N_0$ value of 1 dB are given by

$$\begin{aligned} L_c\boldsymbol{y} \;=\; & ((+0.42, +0.50), (+0.81, -3.93), (-1.73, -2.82), (+3.82, +3.97), \\ & (+1.30, +6.03), (-0.80, -3.30), (-1.32, +6.78), (-2.99, +0.55)). \end{aligned} \tag{7.2}$$

When we compare the sign of the channel values in (7.2) with (7.1) we recognize transmission errors in the second and third information bit and the first parity bit, respectively. When we apply the channel values in (7.2) to the input of the time-continuous simulation model in SIMULINK we obtain a transient response at the decoder output as shown in Fig. 7.2 a). After the settling time of the decoding network we acquire at the output of the ideal analog decoder

$$\boldsymbol{L}_{dec} = (+0.88, -0.93, +1.46, +4.97, +5.03, -4.94, -4.92, -3.24), \tag{7.3}$$

which leads to the correct decoder decision $\hat{\boldsymbol{u}} = \boldsymbol{u} = (+1, -1, +1, +1, +1, -1, -1, -1)$. The correction of the information bits can also be observed when the L-values at the decoder output as plotted in Fig. 7.2 a) are multiplied with the corresponding information bits. We then obtain a transient response as depicted in Fig. 7.2 b) where all values are positive at the end of the decoding, i.e., the sign of the L-values coincides with the sign of the information bits.

### 7.1.2 Switching between Code Words

The dynamic behavior is further analyzed by switching between the two code words

$$\boldsymbol{c}_a = ((+1,+1),(+1,+1),(+1,+1),(+1,+1),(+1,+1),(+1,+1),(+1,+1),(+1,+1))$$

and

$$\boldsymbol{c}_b = ((-1,+1),(-1,+1),(-1,+1),(-1,+1),(-1,+1),(-1,+1),(-1,+1),(-1,+1)),$$

which correspond to the information words

$$\begin{aligned} \boldsymbol{u}_a &= (+1,+1,+1,+1,+1,+1,+1,+1) \quad \text{and} \\ \boldsymbol{u}_b &= (-1,-1,-1,-1,-1,-1,-1,-1), \end{aligned} \tag{7.4}$$

respectively. These two code words are particularly interesting since all outputs of the decoder are required to switch polarity. Both of our chip implementations use 16 single-ended input voltages in combination with a common reference voltage $V_{ref}$. This allows the reduction of the number of input pins on the chip compared to the use of fully differential inputs. The channel values are then represented by the difference between the single-ended input voltages and the reference voltage. Updating the input of the decoder is typically achieved by modifying the 16 single-ended input voltages. For this particular input configuration we use a trick which significantly simplifies the measurement setup for the high-speed chip test. Instead of changing a large number of single-ended input voltages, i.e., all eight inputs for the systematic bits, we switch between the two code words $\boldsymbol{c}_a$ and $\boldsymbol{c}_b$ simply by manipulating the common reference voltage $V_{ref}$. For this, the voltage levels for the following decoder simulations and the later measurement setup are selected as follows. The single-ended input voltages for the information bits and the parity bits are set to 1.7 V and 2.0 V, respectively. The reference voltage $V_{ref}$ is generated by a high-speed signal generator which outputs a rectangular waveform with a lower voltage level of 1.6 V and an upper voltage level of 1.8 V. For $V_{ref} = 1.6$ V the channel values for the information bits and the parity bits are represented at the decoder input with $+100$ mV and $+400$ mV, respectively. The sign of the channel values then equals code word $\boldsymbol{c}_a$. When the reference voltage changes from 1.6 V to 1.8 V the channel values for the parity bits are represented with $+200$ mV while the channel values for the information bits are then represented with $-100$ mV. The sign of the corresponding channel values then equals code word $\boldsymbol{c}_b$. In order to demonstrate the correction of a transmission error in the decoder chip a transmission error is introduced in the first bit position of $\boldsymbol{c}_b$, i.e., the first information bit in $\boldsymbol{u}_b$. This is achieved by clamping the corresponding single-ended input also to 2.0 V as it is the case for the parity bits. The switching between these two input configurations with the single bit error in $\boldsymbol{c}_b$ is illustrated in Fig. 7.3 in terms of the corresponding L-values. We assume that $V_T = 26$ mV. Note that the input for the first information bit with the transmission error exhibits the same characteristic as the channel values for the eight parity bits. Here only the magnitude of the inputs changes but not the sign. The channel values for the remaining information bits change only the sign and maintain the magnitude.

When we apply the channel values from Fig. 7.3 to the input of the time-continuous decoder model in SIMULINK we obtain a transient response at the decoder output as shown in Fig. 7.4. All decoder outputs clearly change the sign from one interval to the next so that the transmission error is corrected successfully.

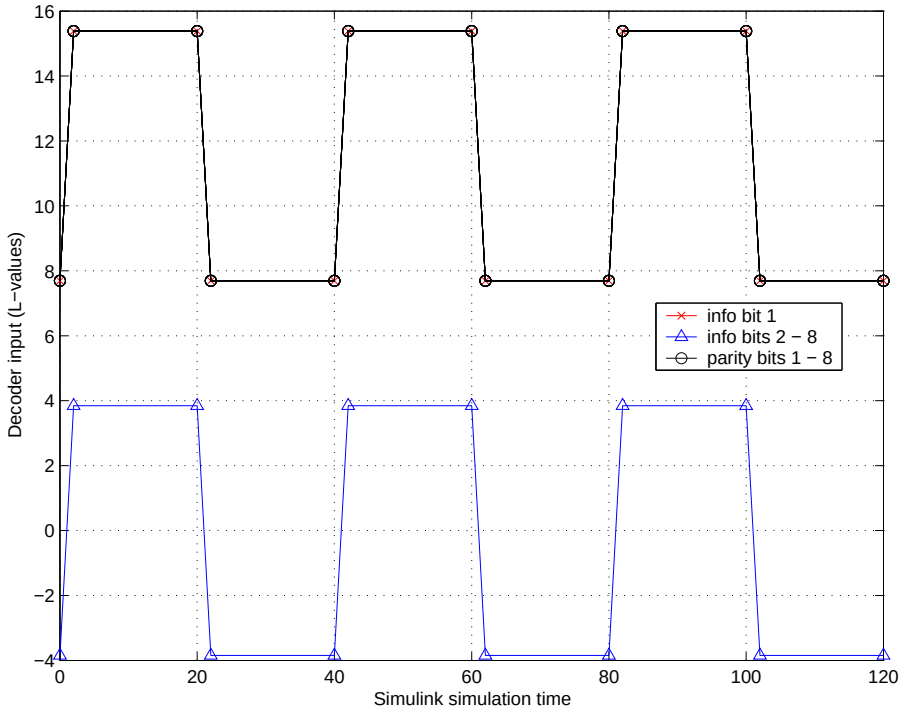



**Figure 7.3**: Switching between the channel values for code word $\boldsymbol{c}_a$ and $\boldsymbol{c}_b$ with a transmission error in the first bit position of $\boldsymbol{c}_b$.

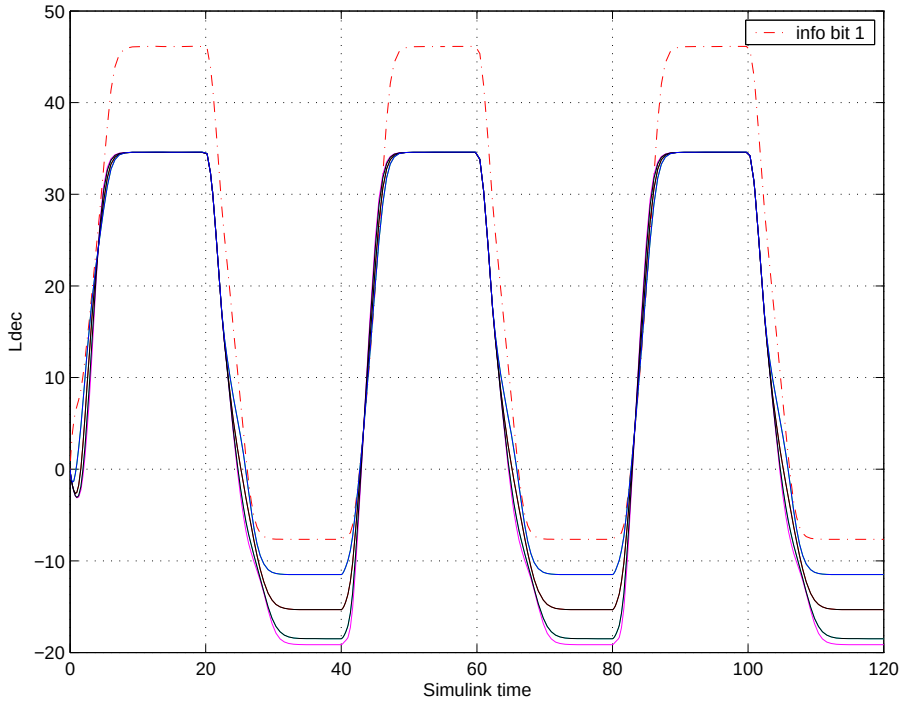



**Figure 7.4**: Transient behavior of the L-values at the output of the ideal analog decoder for the switching input configuration in Fig. 7.3 (transmission error in the first bit position of $\boldsymbol{u}_b$ is corrected).

## 7.2 BiCMOS Decoder Implementation

This section describes our BiCMOS decoder implementation of the tailbiting convolutional decoder in Section 7.1. The shaded region of Fig. 7.5 a) shows the core building block of this decoder implementation that consists of a stacked configuration of emitter coupled transistors, equivalently a Gilbert circuit, connected to diode loads through a connectivity element. The interconnectivity in this element determines the functional response of the block. Balanced nMOS level shifters are used to interface to the core building block. Several Types (I, II and III) of complete building blocks are indicated in Fig. 7.5. The connectivity of Fig. 7.5 b) creates a Boxplus or Type I block, while Fig. 7.5 c) illustrates an adder or Type II/III block. Note that in the Type III block, the lower nMOS current source is removed, see Fig. 7.5 a).

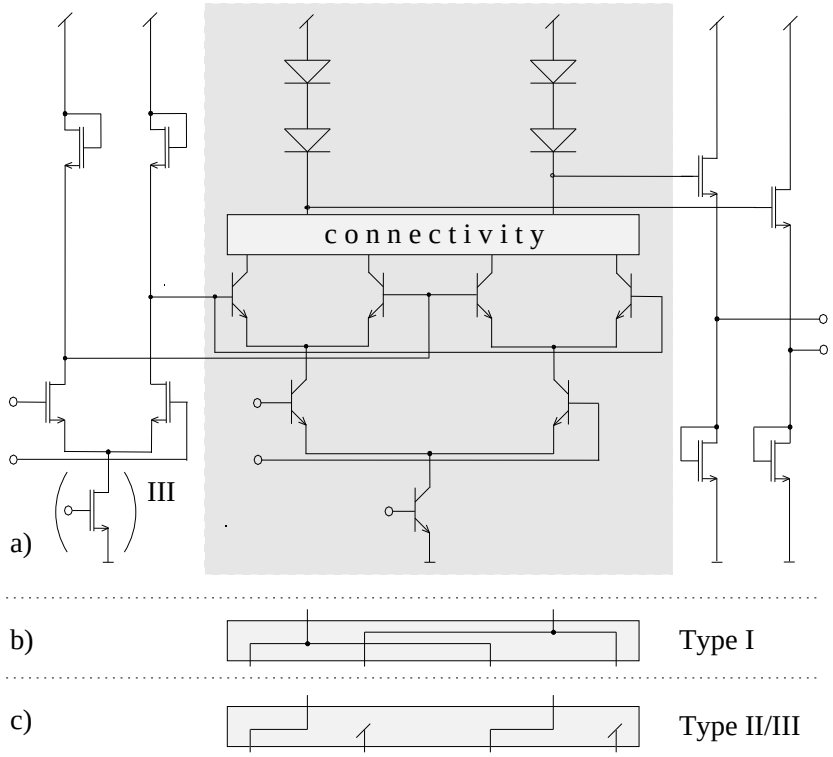



**Figure 7.5**: a) Decoder building block, b) Connectivity for Type I block, c) Connectivity for Type II/III block (III without lower nMOS current source).

The decoding network is illustrated in Fig. 7.6. Soft input values $L_c y_1, \ldots, L_c y_{16}$ are applied to the top of the network and they represent a noisy code word consisting of alternating information and parity bits. Additional a priori information could be used by adding it to the corresponding input values. A tailbiting forward and a backward loop are constructed using the Type I and II blocks. The 16 inputs are applied concurrently to both loops while eight outputs of each loop are combined in a Type III block to generate the soft outputs $L(\hat{U}_1), \ldots, L(\hat{U}_8)$. Thus, input soft values from the channel (and a priori information) can be used directly to generate soft output values in parallel and simultaneously by using both extrinsic and intrinsic information. The sign and magnitude of the soft output value provides the hard decision (digital value) and the reliability of the bit, respectively.

The transient response obtained from circuit-level simulations of the BiCMOS decoder chip for the reference input configuration from Section 7.1.1 is depicted in Fig. 7.7. The differential output voltages represent the a posteriori decoder output $L(\hat{U}_i)$, $i \in \{1, \ldots, 8\}$. Note the similarity between Fig. 7.7 and Fig. 7.2. It can be seen that the last transmission error is corrected

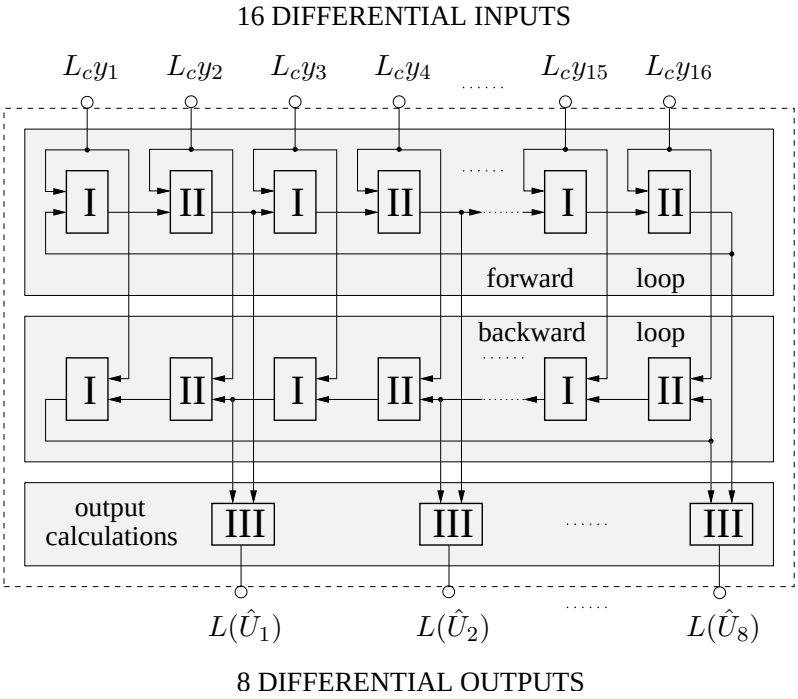



**Figure 7.6**: Block diagram of the BiCMOS decoder chip.

after 50 ns and that the decoder output is settled after around 150 ns.

Fig. 7.8 shows the simulated and measured BER of the chip versus the signal-to-noise ratio. The BER for uncoded transmission is the upper dashed line and is indicated as a reference plot. Two additional simulated sets of curves are plotted: the analog decoder and a conventional MAP decoder. Finally, the measured results of the decoder chip are given. Note that the last three curves overlay one another from 0 to 7 dB. The measurements were performed at room temperature on a HP 8400 test system. A measured transient response of the decoder is shown in Fig. 7.9. Waveform A changes polarity which indicates that a correction has occurred, while waveform B initially changes sign but reverts back to its correct sign. These results agree with simulations. The waveforms are resolved and corrected in less than 20 ns (see decision threshold in Fig. 7.9) and the values are stable at 50 ns. Thus, this decoder will operate robustly at a coded bit rate of 320 Mbps and could potentially support rates approaching 800 Mbps.

The die shown in Fig. 7.18 was fabricated using a 0.25 $\mu$m BiCMOS process. The area of the I/O bound chip is 1.680 mm$^2$ while the active area is approximately 0.120 mm$^2$. The input stage and the core of each module were biased with 80 $\mu$A and 200 $\mu$A, respectively. The total power consumption was measured at 20 mW with a power supply voltage of 3.3 V. The entire decoding network contains 441 npn transistors of minimal size and 356 nMOS transistors. The use of differential circuits throughout the chip reduces temperature and noise effects. Also, power drawn from the supply is constant. Since all modules are biased with the same currents and the power consumption is low the same temperature on the whole chip can be assumed.

An equivalent digital implementation in the probability domain requires 208 multiplication and 64 summation operations per decoding run. A trellis section can be formed in the digital domain using six multipliers and two adders (6x 2+). One instance of this physical component will be used in both the forward and backward loops. Looping back on itself eight times with the appropriate new digital values is equivalent to one pass through the analog path. But to

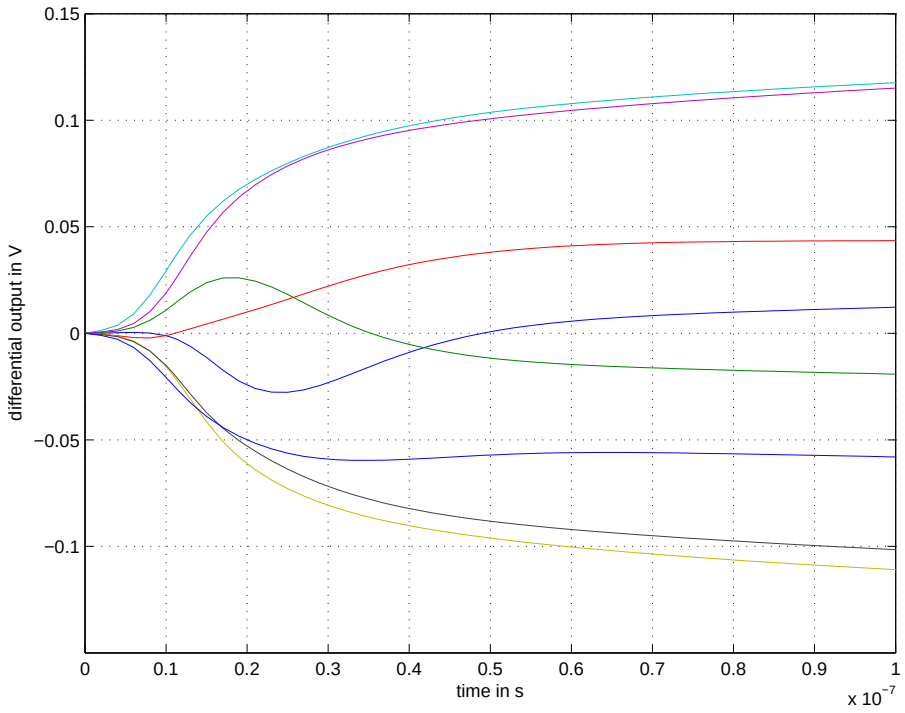



**Figure 7.7**: Circuit simulation of the BiCMOS decoder chip at $0^\circ$ C (reference decoding situation).

achieve a stable output value, the digital implementation needs another eight loops through the component. This digital architecture reduces the area usage of the multipliers and adders at the expense of memory and should be a fair comparison. As illustrated in Table 7.1, the estimated digital period, power and area is compared against the measured results of the analog decoder. If a resolution of 8 bits is assumed, then the digital equivalent decoder would have an overall performance degradation of 3.3x, a power dissipation increase of 8x and an area increase of 5.2x. As is the case for a digital design, one of these parameters can be decreased at the expense of the other two parameters.

**Table 7.1**: Comparison between the measured analog decoder and an equivalent estimated digital implementation.

| type | type of blocks | Raw rate (Mb/sec) | # of occur (#) | time /section (nsec) | total period (nsec) | total power (mW) | Area ($10^{-3}$mm$^2$) | Comments |
|---|---|---|---|---|---|---|---|---|
| **DIGITAL** | | | | | | | | Estimation |
| forward | 6x 2+ | 100 | 1 | 10 | 160 | 70.4 | 189.2 | A/D not included |
| backward | 6x 2+ | 100 | 1 | 10 | 160 | 70.4 | 189.2 | |
| output | 2x | 200 | 8 | 5 | 5 | 18.8 | 246 | |
| aggregate total | | | | | 165 | 159.6 | 624 | |
| **ANALOG** | | 320 | 1 | | 50 | 20 | 120 | Measurement analog input signals |
| **reduction** | | | | | **3.3x** | **8x** | **5.2x** | |

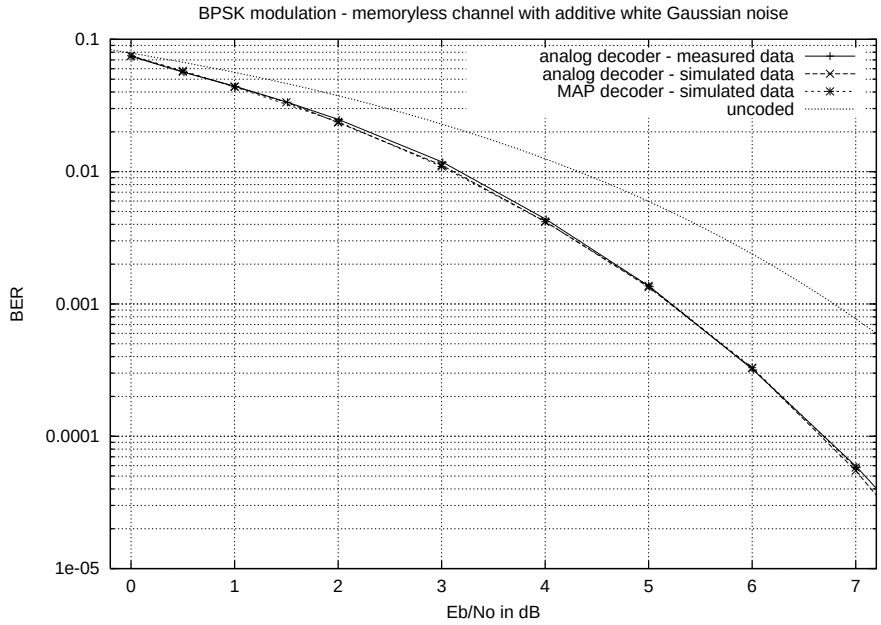



**Figure 7.8**: Simulated and measured BER of the BiCMOS decoder chip.

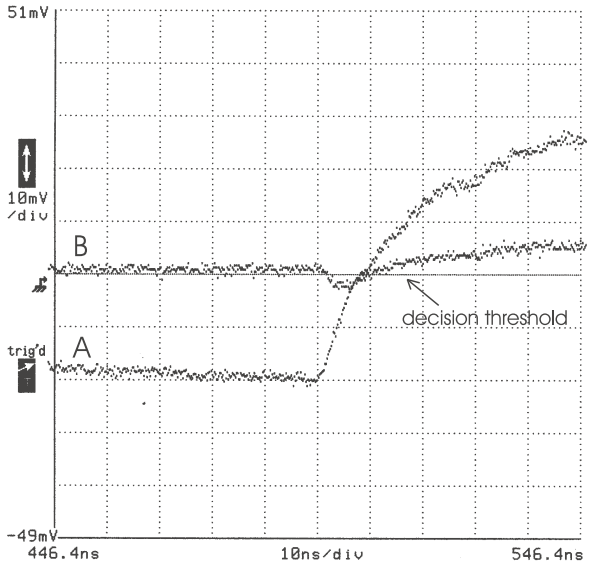



**Figure 7.9**: Measured transient response of the decoding network showing the correction of two bit errors.

## 7.3 SiGe Decoder Implementation

In this section we introduce our SiGe decoder implementation of the tailbiting convolutional decoder in Section 7.1. The core building block of this decoder implementation is shown in Fig. 7.10. Similar to the BiCMOS decoder implementation in Section 7.2 the core consists of a stacked configuration of emitter coupled transistors, equivalently a Gilbert circuit, connected to diode loads through a connectivity element. The main difference compared to Fig. 7.5 is that there is only a single diode load per output. The connectivity element determines the functional response of the block. Two types (A and B) of complete blocks are indicated in Fig. 7.10. Note that Type A and Type B blocks are based on the same connectivity as illustrated in Fig. 7.5 b) and Fig. 7.5 c), respectively. The connectivity of Type A creates a Boxplus block while Type B implements an adder block. The voltage-shifting output stage with load current compensation from Section 6.3.3 is used to interface to the upper inputs of consecutive building blocks.

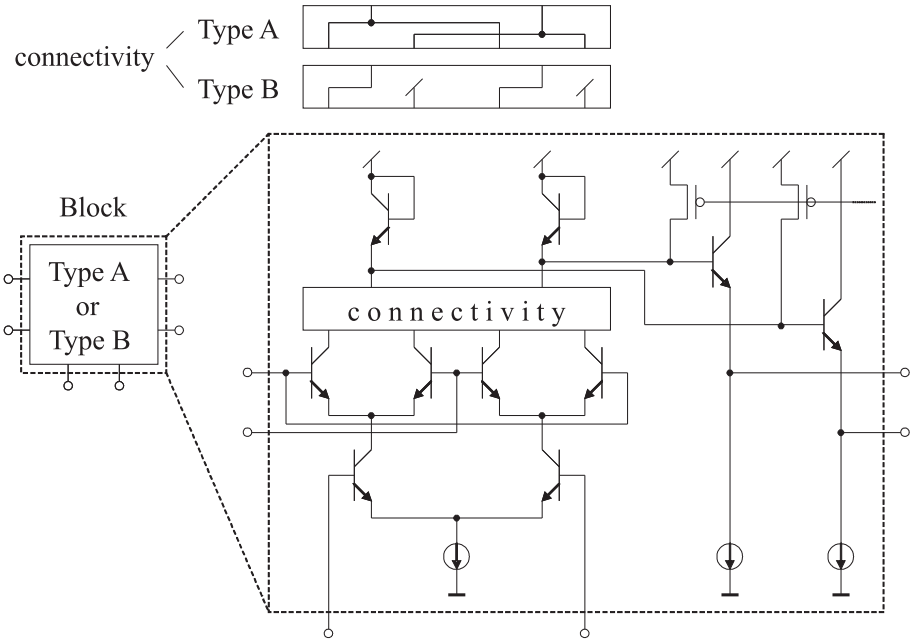



**Figure 7.10**: Decoder building block with connectivity for Type A and Type B blocks.

The decoding network is illustrated in Fig. 7.11. It is consists of eight identical units which are connected together forming a tailbiting forward and backward loop. These units correspond to the individual trellis modules in the decoder. Each trellis module consists of a Type A and Type B block in both the forward and the backward direction as well as an additional Type B block for the computation of the differential output voltage. The inputs for the parity and the information bit are provided as single-ended voltages which are internally converted into differential voltages by Type C blocks. Such a Type C block is depicted in Fig. 7.12. It consists of a differential pair with diode loads where $V_{ref}$ is the common reference voltage for all single-ended input voltages. A resistor is added on top of the diodes in order to lower the output voltage by around 600 mV. The same output stage as in the Type A and B blocks is utilized in order to interface to the lower inputs of consecutive blocks. The Type C* block in Fig. 7.11 is identical to the Type C block but uses a differential input.

Fig. 7.13 shows the transient response obtained from circuit-level simulations of the SiGe decoder chip for the reference input configuration from Section 7.1.1. The differential output voltages represent the a posteriori decoder output $L(\hat{U}_i)$, $i \in \{1, \ldots, 8\}$. Note that the dynamic

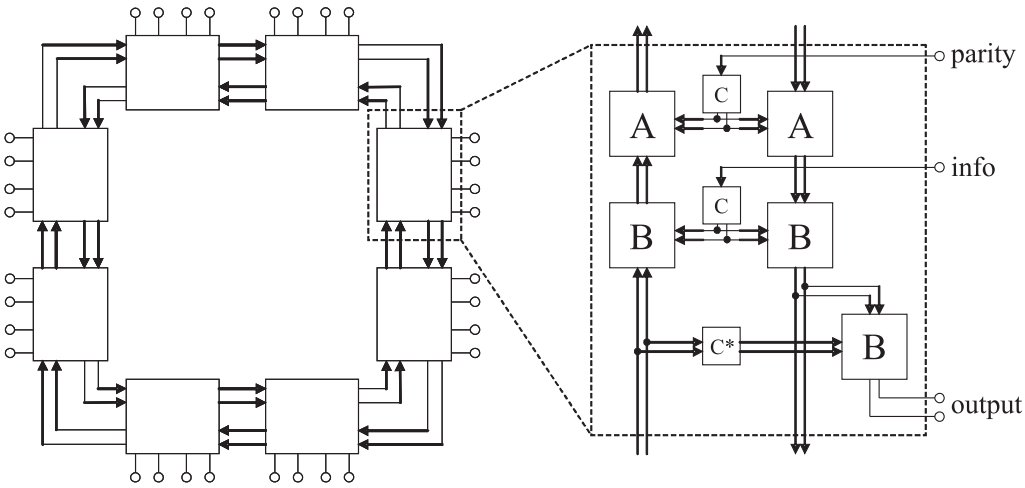



**Figure 7.11**: Block diagram of the SiGe decoder chip.

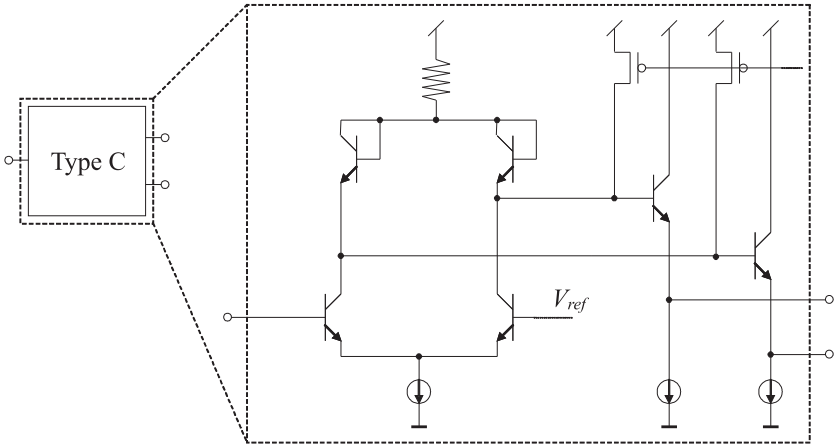



**Figure 7.12**: Type C decoder building block.

behavior in Fig. 7.13 is almost identical to Fig. 7.2. After 540 ps the last transmission error is corrected and after 1.6 ns the decoder output is almost settled.

The circuit-level simulation results for the code word switching as described in Section 7.1.2 are shown in Fig. 7.14. Similar to Fig. 7.4 all outputs change the sign and the introduced transmission error in the first bit position of $\boldsymbol{u}_b$ is corrected successfully. The main difference is that the differential output voltages saturate at a magnitude of around 300 mV. This effect is best observed for positive differential output voltages. Here, the soft output for the first information bit is almost identical to the soft output for the other information bits. The simulation results for the ideal analog decoding network in Fig. 7.4 yield $L(\hat{U}_1) \approx 46$ and equal L-values of around 34.5 for the other information bits. Furthermore, we notice in Fig. 7.14 that even after 100 ns the decoder output is not settled completely. Here, the settling of the decoder output requires at least two orders of magnitude longer compared to the 1.6 ns in Fig. 7.13. The main reason for this is the large magnitude of the differential outputs in Fig. 7.14. This output needs to be provided by Type B blocks, see Fig. 7.11. A large differential output implies that effectively the whole bias current of this block is forced through one diode while the current through the other diode needs to be decreased to almost zero. The latter is extremely time consuming but does not alter the decoder decision. On the other hand, we find that going back to equal currents in both diodes, i.e., achieving a zero differential output voltage, is achieved quickly. In gen-

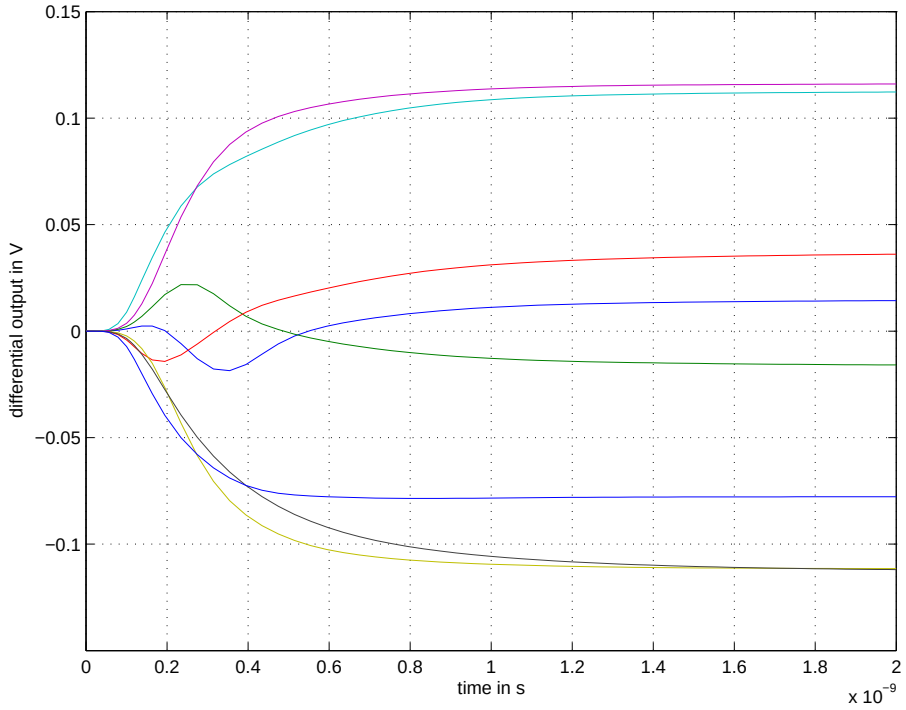



**Figure 7.13**: Circuit-level simulation of the SiGe decoder chip at $25°$ C (reference decoding situation).

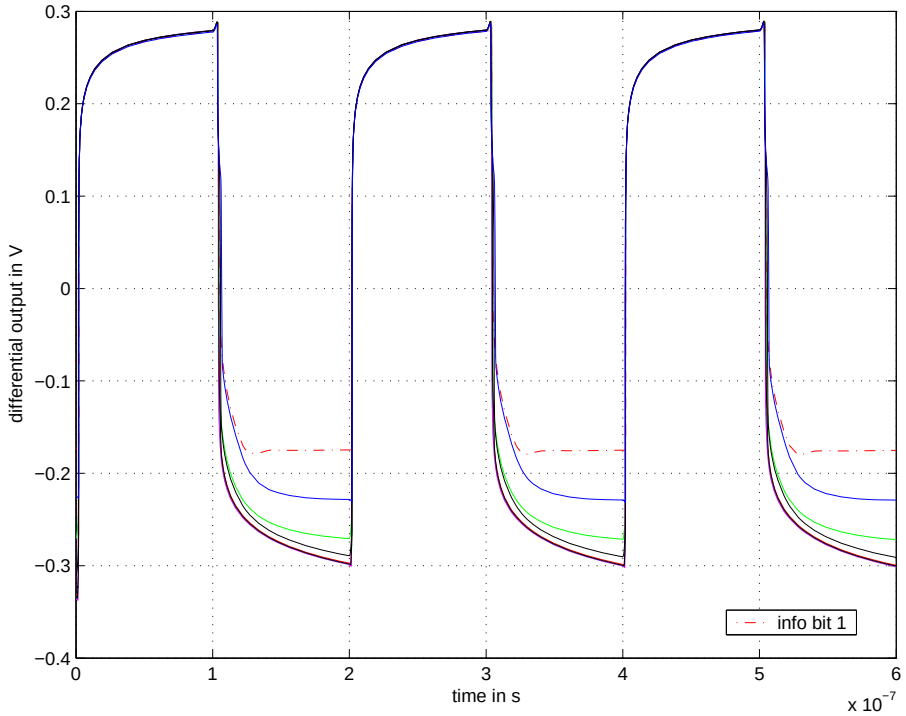



**Figure 7.14**: Circuit-level simulation for the switching input configuration in Fig. 7.3 (transmission error in the first bit position of $\boldsymbol{u}_b$ is corrected).

eral, the decoder reaches its maximum speed around the decision threshold where all transistors operate with relatively large currents. Note that this effect is not appropriately reflected in the time-continuous simulation model with the lumped RC elements. Hard decisions can thus be obtained very fast while extremely reliable bit decisions are particularly time-demanding. We can then argue that the settling of the analog decoder occurs in general faster towards lower SNR values since the magnitude of the decoder output, i.e., the reliability, is typically smaller.

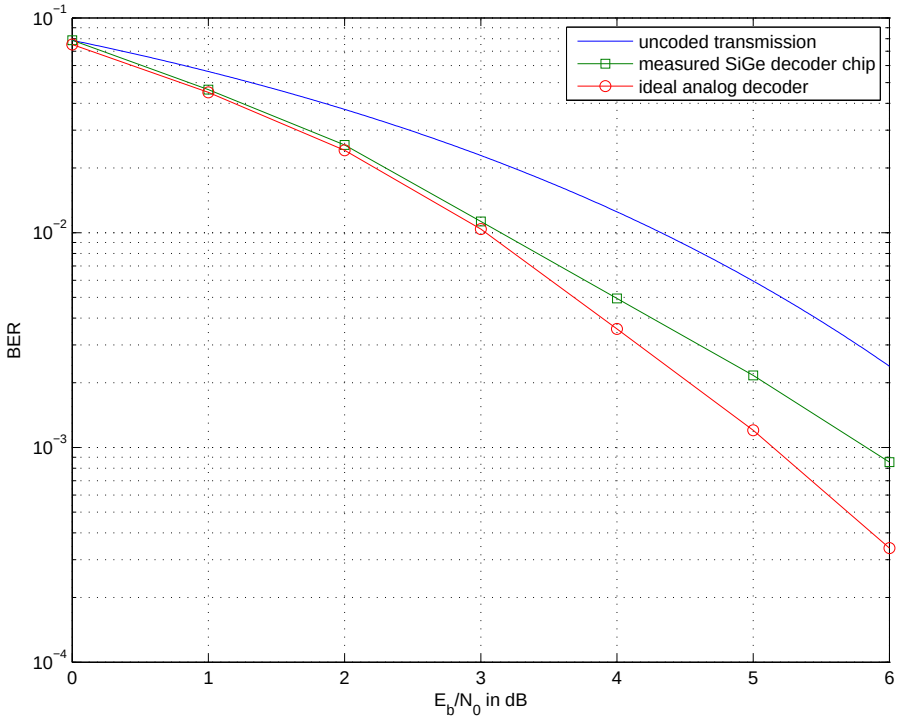



**Figure 7.15**: Simulated and measured BER of the SiGe decoder chip.

A packaged version of the decoder chip was soldered on a test board and then tested at room temperature. The measurement setup included two laptop computers equipped with National Instruments D/A and A/D converters in the PC card slots. The first laptop generated the 16 single-ended input voltages using two DAQ cards 6715. It then triggered the second laptop to measure the eight differential output voltages using a DAQ card 6062E and to evaluate the measurement results. Fig. 7.15 depicts the simulated and measured BER of the SiGe decoder chip. The measurement results nicely approximate the performance of the ideal analog decoder up to a $E_b/N_0$ value of 3 dB. When the SNR is increased further we recognize an increasing gap between the measured and the simulated BER values. At the lowest measured BER we observe an offset of 0.75 dB. This offset may increase further at larger SNR values. Lower BER values could not be measured due to a loss of synchronization between the two laptop computers whenever a large number of voltage vectors was acquired. In the following we try to identify the cause for this suboptimal behavior towards lower BER values. We noticed in the measurement setup of the decoder chip that all-zero inputs (in terms of L-values) did not generate all-zero outputs as expected. For this test, we generated input voltages with the DAQ cards which were equal to the reference voltage $V_{ref}$. We then measured the differential output voltages we found signals with a magnitude of as large as 10 mV. Such an offset at the output shifts the decision threshold and thus leads to an increased BER. We tried to calibrate the measurement setup by subtracting the measured offset voltages from the decoder outputs, but this did not improve the BER performance. A closer analysis of the measured BER values

reveals that individual measured outputs produce a significantly different number of bit errors. The number of bit errors for each bit position of the decoder output is listed in Table 7.2 for the relevant $E_b/N_0$ values of 4 dB, 5 dB and 6 dB. Despite the small number of overall bit errors we can clearly recognize that the most bit errors occur for information bits $u_3$ and $u_4$. This effect appears to be independent of the SNR and is most prominent for a $E_b/N_0$ value of 6 dB where the gap between the simulated and the measured BER curves is large. Here, we count up to 5.5 times the number of bit errors for these bit positions compared with the bit position with the smallest number of bit errors. The distribution of the bit errors at the output of the ideal analog decoder in Fig. 7.15 is listed in Table 7.3 for comparison. Given the small number of samples this distribution can be considered as a reasonable good approximation of a uniform distribution.

**Table 7.2**: Distribution of measured bit errors at different SNR values.

| $E_b/N_0$ | # of bit errors | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | bit position | | | | | | | | total | average |
| | $u_1$ | $u_2$ | $u_3$ | $u_4$ | $u_5$ | $u_6$ | $u_7$ | $u_8$ | | |
| 4 dB | 17 | 18 | 30 | 30 | 19 | 19 | 20 | 14 | 167 | 20.9 |
| 5 dB | 10 | 30 | 49 | 32 | 27 | 27 | 25 | 15 | 215 | 26.9 |
| 6 dB | 21 | 39 | 71 | 58 | 51 | 18 | 13 | 13 | 284 | 35.5 |

**Table 7.3**: Distribution of bit errors at the output of an ideal analog decoder at a $E_b/N_0$ value of 6 dB.

| $E_b/N_0$ | # of bit errors | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | bit position | | | | | | | | total | average |
| | $u_1$ | $u_2$ | $u_3$ | $u_4$ | $u_5$ | $u_6$ | $u_7$ | $u_8$ | | |
| 6 dB | 15 | 15 | 14 | 17 | 14 | 15 | 14 | 9 | 113 | 14.1 |

Fig. 7.16 depicts the measured BER for the individual bit positions in comparison with the (average) measured BER of the decoder chip. Note that due to the small number of bit error for the individual bit positions the corresponding BER can only be considered as a rough approximation of the true BER values, i.e., the plotted values exhibit a large variance. However, Fig. 7.16 also clearly demonstrates that the increased BER of the measured decoder chip is dominated by the outputs for information bits $u_3$ and $u_4$. Given the fact that only a small number of decoder outputs degrade the overall BER performance makes it to appear unlikely that the suboptimal performance is design or the fabrication related. It is important to emphasis that each decoder output originates from essentially identical trellis modules. The layout of such a trellis module was generated only once and then simply replicated on chip. Furthermore, the use of differential circuits throughout the chip reduces temperature and noise effects. Also, power drawn form the supply is constant. Since all modules are biased with the same currents and the power consumption is rather low the same temperature on the whole chip can be assumed. We therefore conclude that the suboptimal decoder performance originates in the measurement setup. The offset of the differential output voltages for all-zero inputs (in terms of L-values) may be caused by various effects. This includes the soldering joint between the packaged chip an the test board, the wiring between the test board and the DAQ cards including intermediate screw connectors as well as the calibration of the DAQ cards. However, the ultimate cause could not be identified.

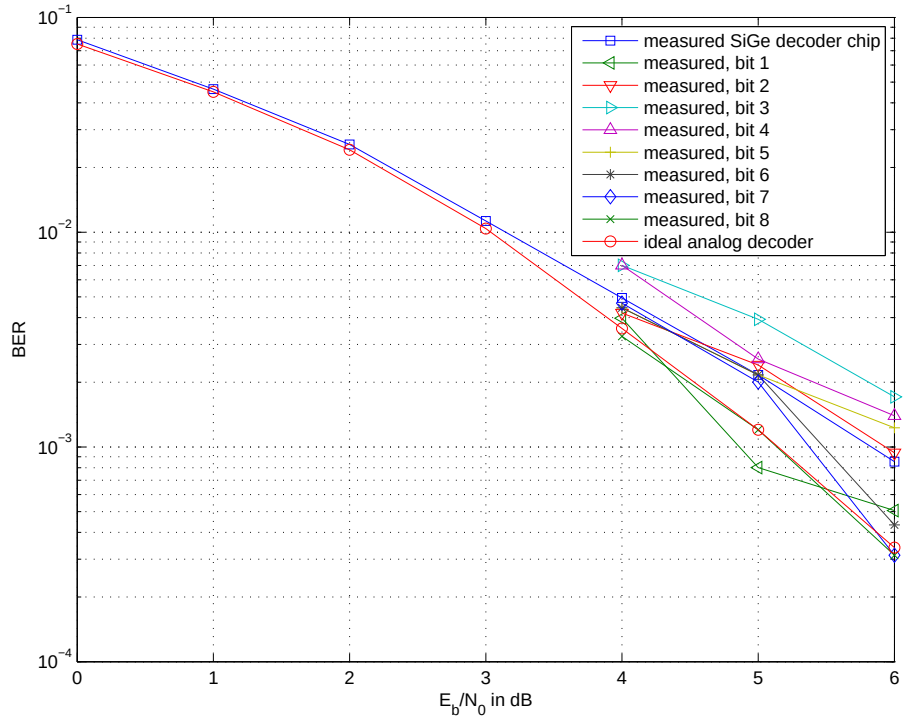



**Figure 7.16**: Measured BER of the SiGe decoder chip together with the measured BER for the individual bit positions.

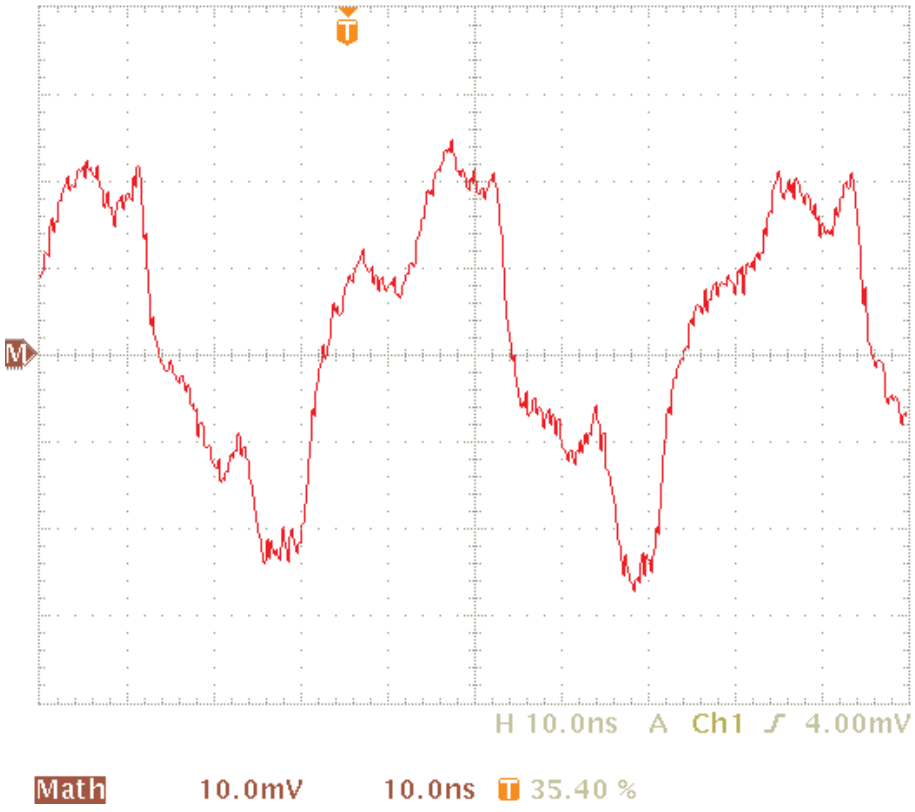



**Figure 7.17**: Transient response of the decoder output for $u_1$ for the switching between code words $\boldsymbol{c}_a$ and $\boldsymbol{c}_b$ with a transmission error in $u_1$.

Further measurements included the switching between the two code words $\boldsymbol{c}_a$ and $\boldsymbol{c}_b$ where a transmission error is assumed in the first bit position of $\boldsymbol{c}_b$, i.e., the first bit position of $\boldsymbol{u}_b$, as described in Section 7.1.2. For this, the single-ended input voltages were again generated with the two DAQ cards and then kept constant during the measurements. The switching between the two input configurations was then realized by means of a 25 MHz signal generator which applied an approximately rectangular pulse to the input for the reference voltage $V_{ref}$. This setup allowed us to switch every 20 ns from one input configuration to the other according to Fig. 7.3. The measured differential output voltage for the first information bit, i.e., the bit with a transmission error, is depicted in Fig. 7.17. It is important to note that the test board was not appropriately shielded during these measurements. Despite the heavy distortion of the measured output we find that the output reliably switches between a positive and a negative voltage as predicted by Fig. 7.14. This indicates that the information bit is decoded correctly for both code words despite the introduced transmission error. Note that the required time for changing the polarity of the differential output voltage including error correction is only in the range of 3-4 ns so that the decoder chip could potentially support coded bit rates approaching 4 Gbps. When we allow 10 ns for decoding a code word the decoder reaches a coded bit rate of 1.6 Gbps. In this context it should be mentioned that this particular test setup has not been considered during the design process. The input for the reference voltage $V_{ref}$ was assumed to carry a DC input signal instead of a high frequency signal. The corresponding wire was therefore implemented in the layout as a large ring within the decoder chip which lies above a ground plate. This introduces a significant capacitance which limits the speed of the transient response in this particular measurement setup. However, this effect is considered to give reasonable ample margin for our speed estimations in the above.

The die shown in Fig. 7.18 was fabricated using the IBM6HP 0.25 $\mu$m SiGe process. The area of the I/O bound chip is 1.638 mm$^2$ while the decoder occupies only 0.53 mm$^2$. The chip has been packaged in a 40 lead, 6 mm x 6 mm Amkor Micro Leadframe (MLF) package. The core and output stage of each block were biased with 400 $\mu$A and 100 $\mu$A, respectively. The total power consumption was measured at 127 mW with a power supply voltage of 3.3 V. The entire decoding network contains 611 npn transistors of minimal size, 129 pMOS, 130 nMOS transistors and 24 resistors.

## 7.4 Summary of Key Parameters

**Table 7.4**: Comparison of our BiCMOS and SiGe decoder chips.

| | 1st prototype | 2nd prototype |
|---|---|---|
| year of fabrication | 1999 | 2001 |
| code | (16,8,3) tailbiting convolutional code | |
| process | 0.25 $\mu$m BiCMOS | 0.25 $\mu$m SiGe |
| overall chip area | 1.680 mm$^2$ | 1.638 mm$^2$ |
| # of transistors | | |
| - npn | 441 | 611 |
| - nMOS | 356 | 130 |
| - pMOS | - | 129 |
| - total | 797 | 870 |
| power supply voltage | 3.3 V | 3.3 V |
| bias currents | | |
| - input stage | 80 $\mu$A | - |
| - core | 200 $\mu$A | 400 $\mu$A |
| - output stage | - | 100 $\mu$A |
| measured power consumption | 20 mW | 127 mW |
| throughput (info bits) | | |
| - estimated | 160 Mbps | 800 Mbps |
| - potentially | 400 Mbps | 2 Gbps |
| energy per info bit | | |
| - estimated | 0.125 nJ/bit | 0.159 nJ/bit |
| - potentially | 0.050 nJ/bit | 0.064 nJ/bit |

The key parameters of our analog decoder implementations are summarized in Table 7.4. The two decoder chips allow a direct comparison since they are for the same (16,8,3) tailbiting convolutional code with memory one. Given the limitations of the available measurement equipment we were not able to determine the BER at full decoder speed. Instead, the measured transient behavior of the chips was used in order to estimate the speed of the decoders. The measured transient response of the decoder chips indicates that the BiCMOS decoder easily handles a throughput of 160 Mbps in terms of information bits while the SiGe decoder achieves 800 Mbps. Potentially, the decoder chips support data rates approaching 400 Mbps and 2 Gbps, respectively. When we compare the circuit-level simulation results in Fig. 7.13 with the results in Fig. 7.7 we could expect that the SiGe decoder is around 90 times faster than the BiCMOS decoder. This is true for both the correction of the last bit error as well as the settling of the decoder outputs. However, based on the measured transient response it is estimated that the SiGe decoder chip operates only five times faster than the BiCMOS decoder. This may be caused by the large capacitance of the $V_{ref}$ input and the measurement setup with the packaged chip soldered on a test board. The decoder chips exhibit almost an identical chip area and have comparable transistor count. Both use a supply voltage of 3.3 V. The power consumption of the SiGe decoder is with 127 mW around six times higher than in the BiCMOS decoder. The estimated energy per information bit is 0.125 nJ/bit for the BiCMOS decoder and 0.159 nJ/bit for the SiGe decoder. Die micrographs of our two decoder chips are shown in Fig. 7.18 and Fig. 7.19.

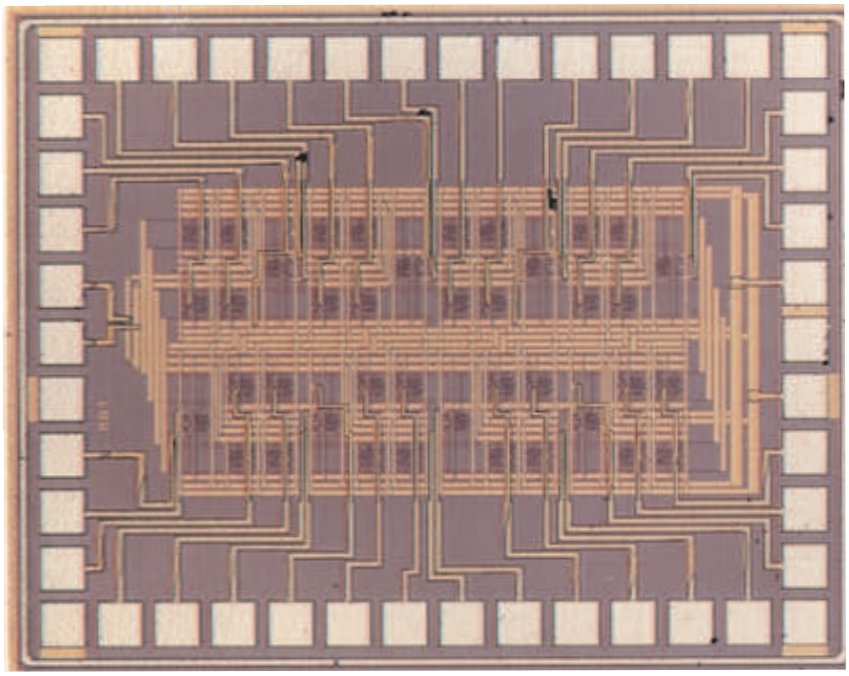


**Figure 7.18**: Die micrograph of our BiCMOS decoder implementation.

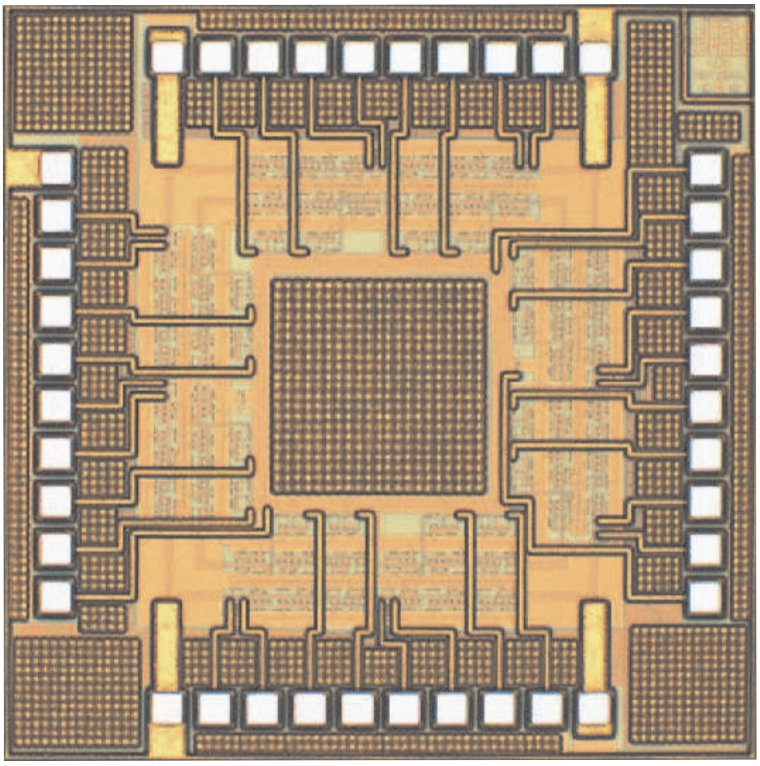


**Figure 7.19**: Die micrograph of our SiGe decoder implementation.

# 8

# Outlook to Real World Applications - Example IEEE 802.11n

One of the key questions in analog decoding is whether it can be applied to real world applications and what performance gain can be expected in terms of speed, area and power consumption compared to digital decoder implementations. At present no conclusive answers are available. The intension of this chapter is to highlight these questions and also to identify areas for further study. In the following we investigate analog LDPC decoders for the emerging new standard IEEE 802.11n [IEE05] for wireless LANs. Our investigations are based on the fully parallel analog LDPC decoder architecture as introduced in Section 5.4. We restrict ourselves to the $(N, K) = (648, 324)$ and $(N, K) = (648, 540)$ LDPC codes with code rates of $R = 1/2$ and $R = 5/6$, respectively. The BER performance of these two coding schemes is shown in Fig. 5.36 and the effect of quantized input values is demonstrated in Fig. 5.37. We examine an implementation of these two analog LDPC decoders in the 0.18 $\mu$m CMOS technology as offered by the Taiwan Semiconductor Manufacturing Company (TSMC). The supply voltage is 1.8 V. The Boxplus (BPX) and summation (SUM) building blocks from Sections 6.1.5 and 6.1.6 are here implemented with nMOS transistors. The voltage-shifting output stage from Section 6.3.2 is utilized in order to interconnect the blocks. The CMOS implementation of such complex analog LDPC decoders poses two major challenges. First, the BPX and SUM building blocks implemented in CMOS are suboptimal. This is due to the non-exponential behavior of nMOS transistors as outlined in Section 6.1.1. It is therefore not clear to what extend this impacts the performance of such a large analog decoder implementation at system level. This directly leads to the second big challenge, the large size of these LDPC decoders. This not only causes severe problems for the simulation of these decoders but also for the physical layout because of routing complexity. Furthermore, real world applications typically require that the analog decoder is integrated into a digital receiver where the decoder input is provided in the form of quantized soft information and the decoder outputs hard decisions. This motivates the investigation of digital input and output interfaces with the associated D/A and A/D converters, respectively. These converters are commonly assumed to be critical in terms of area and power consumption.

We introduce the different components of an analog LDPC decoder including the input and output interfaces in Section 8.1. We then estimate the BER performance of CMOS implemen-

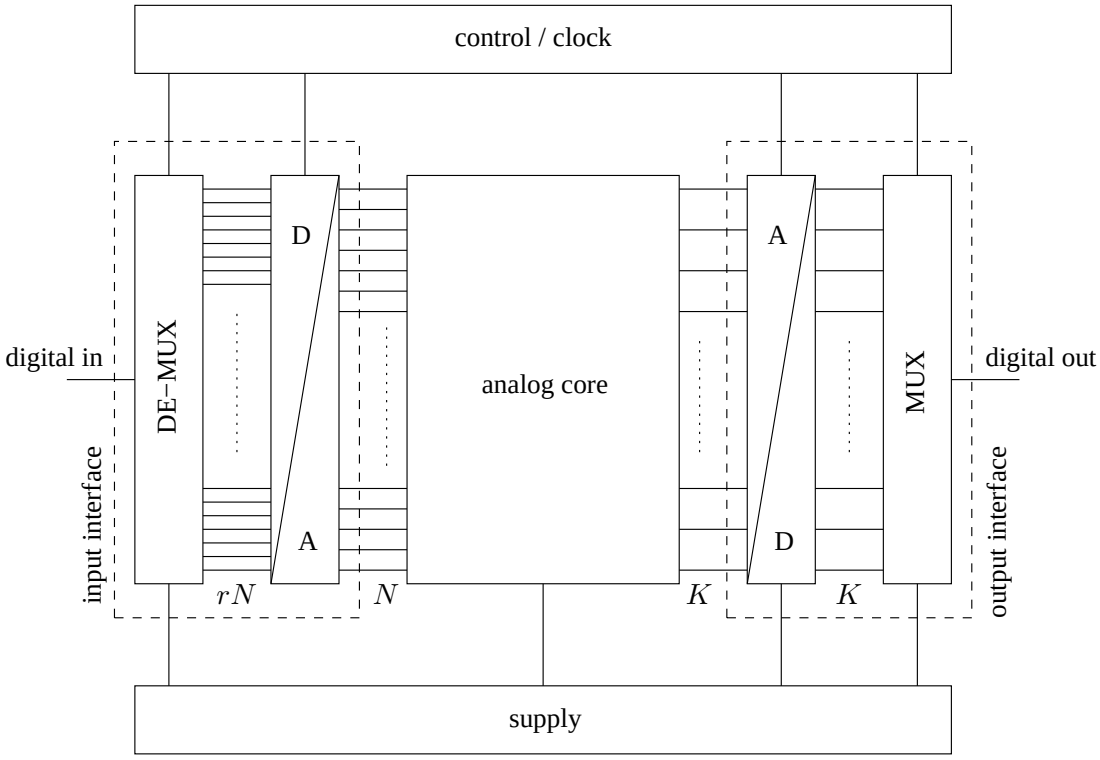



**Figure 8.1**: Block diagram of the analog LDPC decoder with digital input and output interfaces.

tations in Section 8.2 and investigate various possible impairments in Section 8.3. This chapter concludes with a speed estimation of the LDPC decoders and an estimation of decoder performance in terms of transistor count, chip area and power consumption.

## 8.1 Block Diagram of an Analog LDPC Decoder

The block diagram of the complete LDPC decoder is shown in Fig. 8.1. At the input of the decoder there is a digital input interface which receives quantized soft information. We demonstrated in Section 5.4.3 that a quantization of the channel values with $r = 3$ and $r = 4$ bits already yields satisfactory results, see Fig. 5.37. With a quantization of three bit there is a loss of 0.12 dB and 0.1 dB for the rate $R = 1/2$ and $R = 5/6$ codes, respectively. In case of a quantization with four bits the loss reduces to less than 0.05 dB in both cases. The incoming bit stream is de-multiplexed and then converted into analog inputs for the LDPC decoder core. The decoder core processes the overall block length at the same time. The soft output of the decoder core is then converted into hard decisions. The decoded information bits are then multiplexed and provided as sequential bit stream at the output of the overall decoder. The additional circuitry for controlling the input and output interfaces including the provision of clock signals is neglected in our estimations. In the following sections we investigate the different blocks of the analog LDPC decoder in Fig. 8.1 in more detail.

Digital input and output interfaces of analog decoders are, e.g., also investigated in [HLL$^+$99] and [Lus00].

### 8.1.1 Decoder Core

The decoder core consists of a fully parallel analog LDPC decoder as it has been introduced in Section 5.4.3. The circuit implementation of the individual check node and variable node pro-

cessors in the decoder has been covered in Section 6.4.1. In this section we are more interested in the complexity of the analog decoder core and the resulting challenges for the physical layout of the decoder. We start with a complexity analysis in terms of the required number of node processors and transistors. In Section 6.4.3 we determined for the required number of check node processors (CNP) and variable node processors (VNP)

$$\#\text{CNP} = c_H - 2(N - K) \tag{8.1}$$

and

$$\#\text{VNP} = c_H - N, \tag{8.2}$$

respectively. Here, the parameter $c_H$ determines the number of ones in the corresponding $(N - K) \times N$ parity-check matrix of the code. The LDPC codes as proposed in [IEE05] were designed so that both the rate $R = 1/2$ and the rate $R = 5/6$ code exhibit $c_H = 2376$. The resulting number of check node and variable node processors in the corresponding analog decoder cores are summarized in Table 8.1 together with the overall number of nMOS transistors. We

**Table 8.1**: Complexity of analog LDPC decoder cores in terms of node processors and nMOS transistors.

| | LDPC decoder core | |
|---|---|---|
| $(N, K)$ code | (648,324) | (648,540) |
| code rate | $R = 1/2$ | $R = 5/6$ |
| #CND | 1728 | 2160 |
| #VND | 1728 | 1728 |
| # of nMOS transistors per node processor (from Table 6.2) | 39 | 39 |
| overall # of nMOS transistors | 134784 | 151632 |

assume that 39 nMOS transistors are required for the implementation of a node processor, see Table 6.2. The large number of node processors poses one of the major challenges in the design process of the decoder core. This is because a manual placement of the individual node processors and routing them by hand is clearly not practical. There are commercial tools available for place and route which mainly originate from digital design environments. However, these tools are not well adopted to such complex tasks in the analog domain where a large number of differential signal wires needs to be routed. In the following we outline a possible solution for this design challenge which allows us to significantly reduce the complexity of the place and route task. We exploit the structure of the proposed LDPC codes and develop a hierarchical approach for the layout based on the structure of the parity-check matrices. The top-level of such a hierarchy in the layout is depicted in Fig. 8.2 and Fig. 8.3 for the rate $R = 1/2$ and $R = 5/6$ codes, respectively. Each rectangle represents a layout module for a $z \times z$ sub-matrix of the corresponding parity-check matrix. Given the structure of these LDPC codes each of the $z \times z$ matrices represents a cyclically shifted version of the identity matrix. For the considered block lengths of the codes we have $z = 27$. It is then sufficient to design only one of these layout modules, e.g., the one for the identity matrix as indicated in Fig. 8.4, and to guarantee the cyclic shift by an external wiring network. The elements of this layout module are the check node and variable node processors from Section 6.4.1. The wiring of the layout modules can then simply be achieved with analog busses which propagate in horizontal and vertical direction equivalent to the rows and columns of the parity-check matrix. The area usage of the individual LDPC decoder cores in Fig. 8.2 and Fig. 8.3 can be reduced by moving the layout modules together in horizontal direction. This allows us to significantly reduce the area in Fig. 8.2 at the expense of only a small modification of the wiring network. In Fig. 8.3 there is only a marginal area

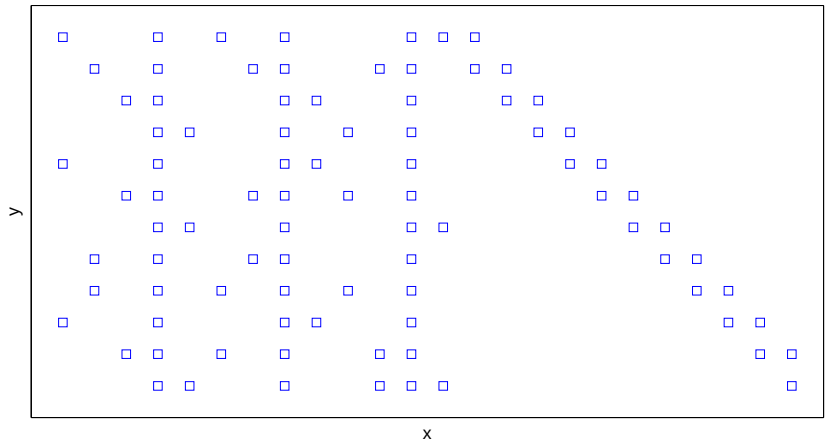



**Figure 8.2**: Top-level layout of the (648,324) LDPC decoder core based on the parity-check matrix (without area optimization).

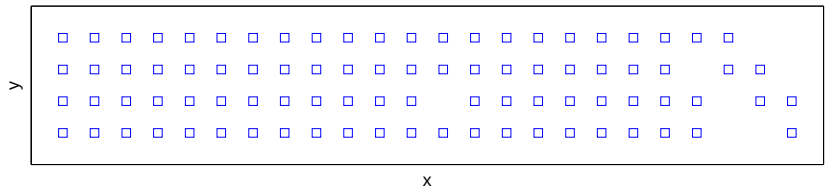



**Figure 8.3**: Top-level layout of the (648,540) LDPC decoder core based on the parity-check matrix (without area optimization).

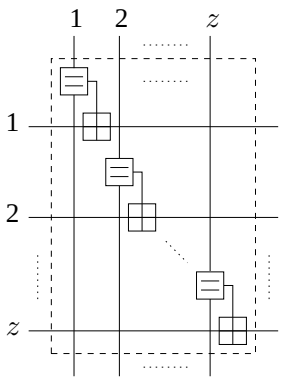



**Figure 8.4**: Block diagram of a layout module in Fig. 8.2 and Fig. 8.3.

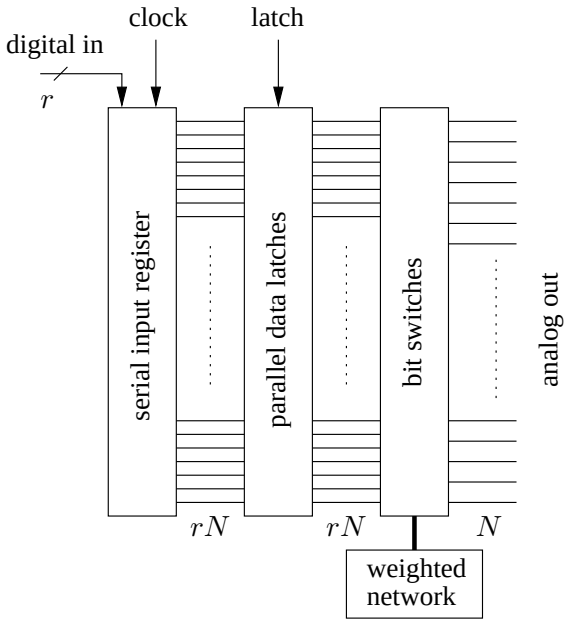



**Figure 8.5**: Digital input interface of the analog LDPC decoder including D/A conversion.

reduction possible due to the given density of the layout modules. Note that the first and the last layout module in each row of Fig. 8.2 and Fig. 8.3 require only variable node processors and no check node processors. This is because there are $d_c$ blocks in each row but only $d_c - 2$ check node processors are necessary for the calculation of the internal extrinsic information. Also note that we neglect the fact that no decoder output needs to be provided for the parity bits of the code.

### 8.1.2 Digital Input Interface

The maximum speed of the considered LDPC decoders is specified with 240 Mbps for the rate $R = 1/2$ code and 600 Mbps for the rate $R = 5/6$ code [IEE05]. These data rates refer to uncoded data rates and are achieved through different modulation schemes. Together with a representation of the input values with either $r = 3$ bits or $r = 4$ bits we obtain four different speed requirements for the input interface as listed in Table 8.2. In order to avoid clock signals

**Table 8.2**: Supported data rates at the digital input interface.

| | $R = 1/2$ | $R = 5/6$ |
|---|---|---|
| 3 bit resolution | $3 \cdot 480$ Mbps=1440 Mbps | $3 \cdot 720$ Mbps=2160 Mbps |
| 4 bit resolution | $4 \cdot 480$ Mbps=1920 Mbps | $4 \cdot 720$ Mbps=2880 Mbps |

with several GHz in the receiver we assume that the incoming information is represented by $r$ parallel bit streams running at the speed of the coded bit rate. Each group of $r$ bits coming in simultaneously then represents a quantized channel value for the analog decoder core. The corresponding digital input interface of the decoder is shown in Fig. 8.5. It consists of $r$ serial input registers of length $N$, which, after the overall block of $rN$ bits is loaded, pass on the bits to parallel data latches. The content of the data latches, i.e., $r$ bits for each analog output signal, selects one out of the $2^r$ available voltage values provided by the weighted network in Fig. 8.5. This weighted network is assumed to be adjustable so that different voltage levels can be provided dependent on the channel SNR. The bit switches maintain the analog output for the

duration of the decoding process so that no additional analog memory elements are required. During the decoding process the next block of input bits can be loaded into the serial input register.

### 8.1.3 Digital Output Interface

The digital output interface of the decoder is depicted in Fig. 8.6. It consists of $K$ comparators which make the hard decision based on the differential output voltages provided by the analog decoder core. After the decoding time the decoded bits are copied into parallel data latches and the serial output register, from where the decoded information bits are read out in a sequential fashion.

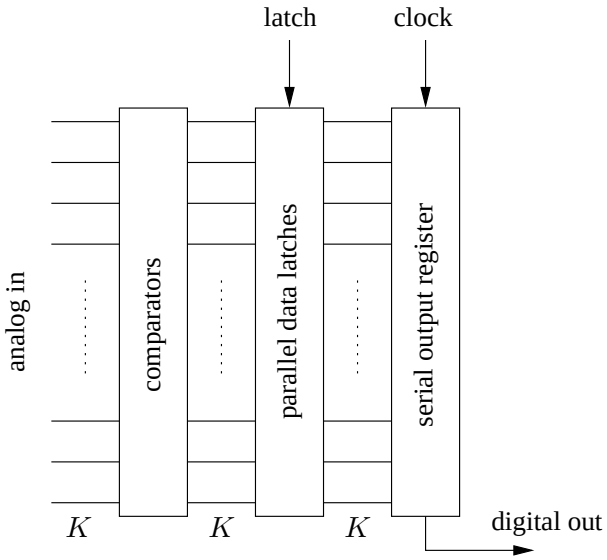



**Figure 8.6**: Digital output interface of the analog LDPC decoder including A/D conversion.

## 8.2 Estimated BER Performance

In general, the BER performance of an analog decoder depends on various different parameters. This includes the technology, the circuit design, the physical dimensions of the transistors, the bias currents used in the building blocks and the temperature. The adequate selection of circuit-level parameters during the design process is critical for the success of a CMOS decoder implementation. The main reason for this is the non-exponential behavior of CMOS devices which only roughly approximates the exponential characteristic of bipolar transistors. In the following, we introduce a new design approach where circuit-level parameters are optimized by means of EXIT charts. We then evaluate the BER performance of analog LDPC decoders for the rate $R = 1/2$ and $R = 5/6$ codes. The simulation results are compared with the BER performance of the corresponding reference decoder as shown in Fig. 5.36. We start with simulations at room temperature (27 °C) and then investigate the impact of temperature variations between -40 °C and +80 °C. We will see that the BER performance of the considered decoders is in general suboptimal, in particular at low temperatures. However, we demonstrate that this performance degradation can be mostly compensated with a special trimming of the decoder.

### 8.2.1 Optimization of Circuit-Level Parameters with EXIT Charts

In this section we optimize the physical dimensions of the nMOS transistors used in the building blocks of the decoder. We utilize a new approach which is based on EXIT chart analysis. The selection of the width W and length L of a CMOS transistor depends mainly on the bias current $I_b$ used for the building blocks. This bias current defines the maximum current through the devices. There is only limited flexibility in the selection of the bias current because it mainly determines the speed of the transistors and thus the speed of the overall decoder. The bias current therefore needs to be selected in a way that the speed requirements of the application are met. In the following we assume a bias current of $I_b = 1$ $\mu$A. The selection of W and L then poses a two-dimensional optimization problem. Our goal is to optimize these parameters based on system-level simulations of the overall LDPC decoder. EXIT charts provide an efficient means for such a performance estimation of a certain decoder configurations so that more time consuming BER simulations can be avoided. Fig. 8.7 depicts the simulated trajectories for the example of the $R = 1/2$ LDPC decoder for different widths of the transistors and a $E_b/N_0$ value of 1.75 dB. The transistors have a length of L=500 nm. We notice that the mutual information

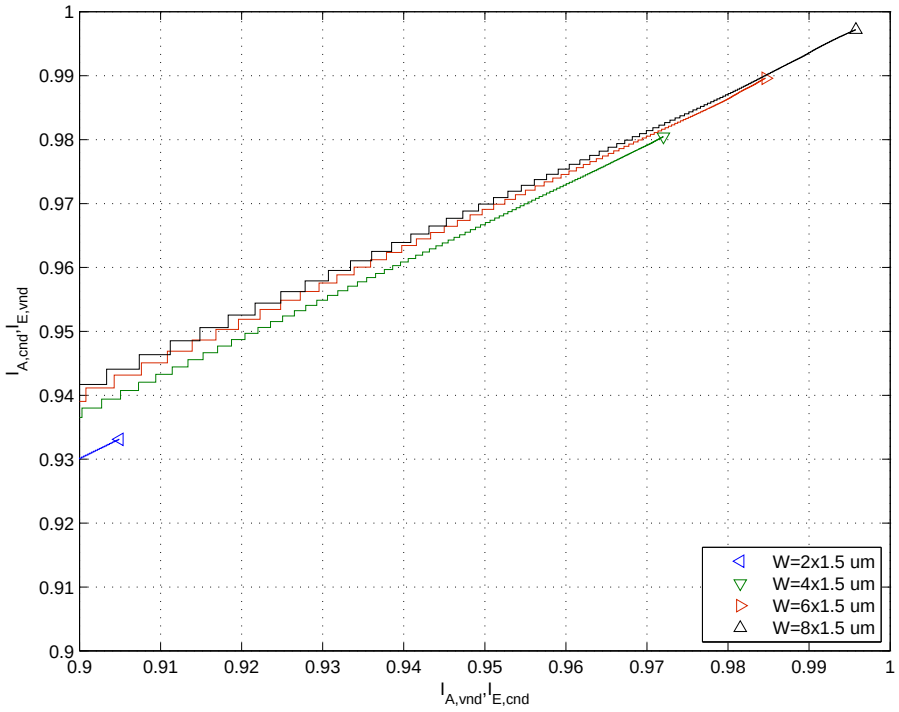



**Figure 8.7**: Trajectories for the $R = 1/2$ decoder with different transistor widths, $I_b = 1$ $\mu$A and a $E_b/N_0$ value of 1.75 dB.

at the end of the decoding process, i.e., at the end of the trajectory, reaches its maximum for the largest considered width of W$= 8 \times 1.5$ $\mu$m. The trajectory for this width closely approaches the upper right corner of the EXIT chart and provides a good trade-off between transistor size and performance. The length L of the transistors can be optimized similarly. This optimization leads to L=500 nm as already assumed in the above. Similar results are obtained for the $R = 5/6$ LDPC decoder so that both decoders can use identical transistors. All of our following investigations are therefore based on a transistor size of W$= 8 \times 1.5$ $\mu$m and L$= 500$ nm with a bias current of $I = 1$ $\mu$A per block.

Other parameters of analog LDPC decoders can also be optimized with EXIT charts.

### 8.2.2 Room Temperature (no Trimming)

We start with a first performance evaluation of the analog LDPC decoders with rate $R = 1/2$ and $R = 5/6$ at room temperature (27 °C). The estimated BER performance of the decoders with the optimized nMOS transistor size from Section 8.2.1 is depicted in Fig. 8.8. We notice that both decoders exhibit a clearly suboptimal performance. Fig. 8.8 also shows the estimated BER performance of these analog LDPC decoders implemented in the SiGe technology which has also been used for our SiGe decoder chip in Section 7.3 ($I_b = 400$ $\mu$A). Note that the SiGe decoders closely approach the performance of the reference decoder and even slightly outperform the reference decoder at a high SNR.

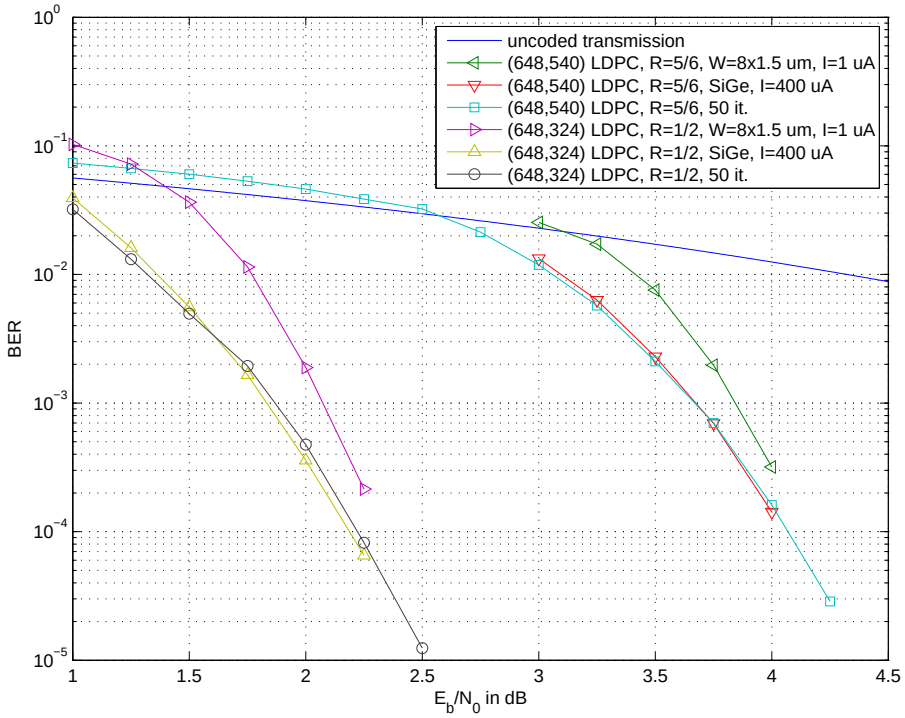



**Figure 8.8**: Estimated BER of the analog LDPC decoders implemented in SiGe and CMOS technology at room temperature (no timming).

### 8.2.3 Temperature Variations (no Trimming)

We now investigate the performance of the CMOS decoders for the case of temperature variations. Fig. 8.9 and Fig. 8.10 show the estimated BER performance of the analog LDPC decoders with rate $R = 1/2$ and $R = 5/6$, respectively. Different temperatures of -40 °C, +27 °C and +80 °C are considered. We notice a severe performance degradation of both decoders at a temperature of -40 °C. The BER performance improves with increasing temperatures. At +80 °C the $R = 1/2$ decoder is slightly better than at +27 °C while the $R = 5/6$ decoder exhibits a comparable performance as at +27 °C. We demonstrate in the next section how this suboptimal decoder performance can be improved.

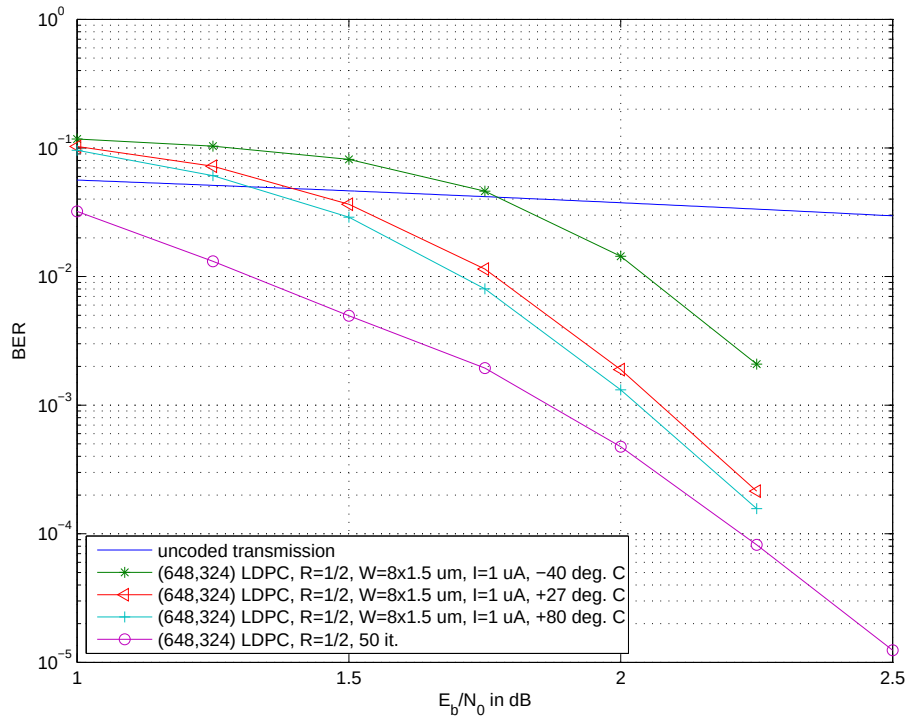



**Figure 8.9**: Estimated BER of the $R = 1/2$ analog LDPC decoder at different temperatures (no trimming).

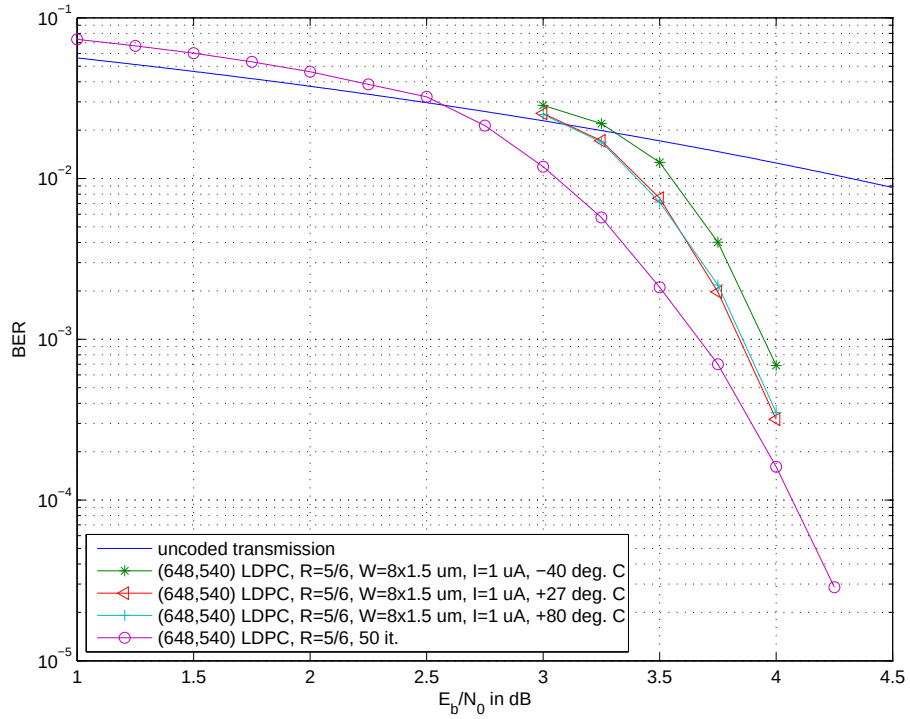



**Figure 8.10**: Estimated BER of the $R = 5/6$ analog LDPC decoder at different temperatures (no trimming).

### 8.2.4 Trimming for Performance Optimization

In Sections 8.2.2 and 8.2.3 it became apparent that the BER performance of analog LDPC decoders implemented in CMOS is clearly suboptimal, compare Fig. 8.9 and Fig. 8.10. Note that this behavior is observed despite the fact that the physical dimensions were already optimized in Section 8.2.1. We now demonstrate how the performance of the CMOS decoders can be optimized in a way that the performance of the reference decoder is closely approached for all temperatures and the full SNR range. This optimization is referred to as trimming of the decoders. So far we always assumed that the differential voltages in the analog circuits represent log-likelihood ratios based on the relation

$$\Delta V = V_T L(X), \tag{8.3}$$

with $V_T$ as the thermal voltage. The expression in (8.3) originates from the description of bipolar transistor circuits, see Section 6.1. For the here considered CMOS implementations we now replace (8.3) with

$$\Delta V = \tilde{V}_T L(X), \tag{8.4}$$

where the factor $\tilde{V}_T$ is subject to further optimization with EXIT charts. Note that this optimization depends on both the temperature and the SNR. The BER simulation results for the rate $R = 1/2$ and $R = 5/6$ analog CMOS decoders after this optimization are depicted in Fig. 8.11 and Fig. 8.12, respectively. Note that both analog CMOS decoders now closely approach the performance of the corresponding reference decoders over the full SNR and temperature range.

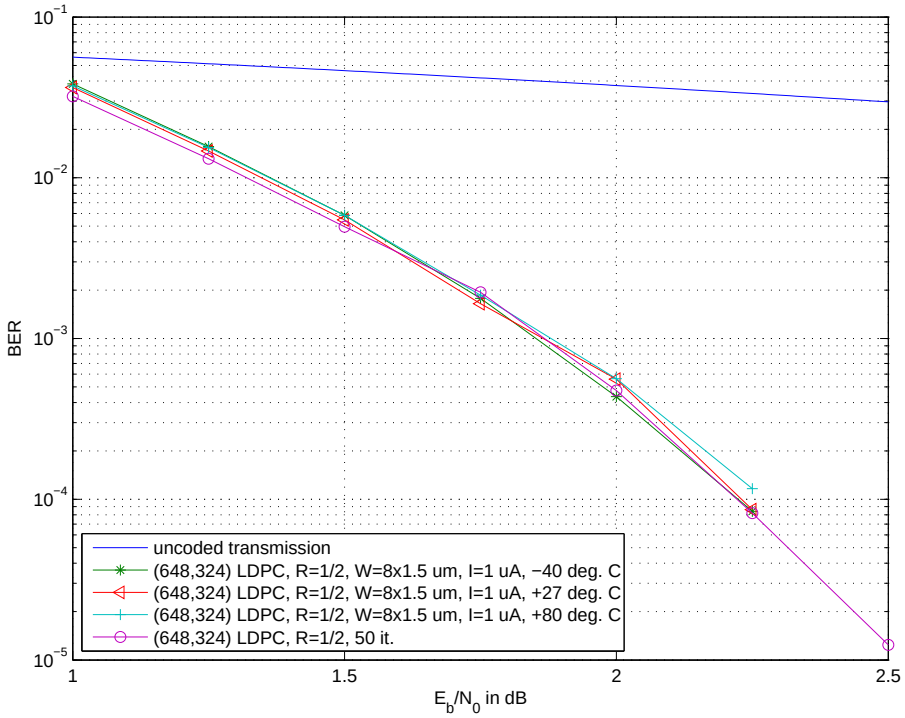



**Figure 8.11**: Estimated BER of the $R = 1/2$ analog LDPC decoder at different temperatures (with trimming).

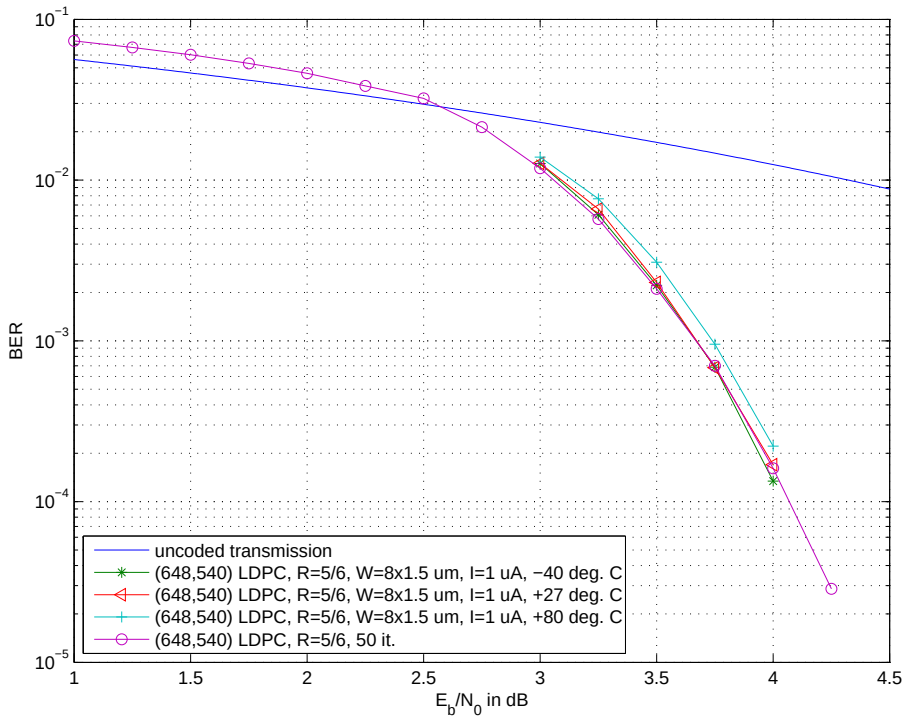



**Figure 8.12**: Estimated BER of the $R = 5/6$ analog LDPC decoder at different temperatures (with trimming).

## 8.3 Impairments of Analog CMOS Decoders

This section focuses on the impairments which are encountered when analog decoders are implemented in CMOS technology. We start with an error analysis of the basic building blocks of an analog LDPC decoder and then turn to noise effects and supply voltage variations. Furthermore, we cover device mismatch, process variance and also give a short outlook to the challenges of further shrinking device sizes.

### 8.3.1 Error Analysis of Decoder Building Blocks

In this section we investigate the error of the two main building blocks of an analog LDPC decoder, namely BPX and SUM. Our analysis is based on the DC behavior of these blocks so that the blocks are fully described in terms of the two log-likelihood ratios $L(X_1)$ and $L(X_2)$ at the input and $L(X_3)$ at the output. The error is evaluated in terms of the relative error according to

$$\text{relative error} = \frac{L_{ideal}(X_3) - L(X_3)}{L_{ideal}(X_3)}, \tag{8.5}$$

where $L_{ideal}(X_3)$ represents the output of the ideal decoder building block. Fig. 8.13 depicts the relative error of the CMOS implementation of the BPX building block as it was used for the simulations in Sections 8.2.2, 8.2.3 and 8.2.4 at room temperature (27 °C). The plot reveals that this block introduces a large error of up to 34 % when the magnitude of both inputs is small. The main reason for this is the non-exponential behavior of the CMOS transistors. When the magnitude of both inputs is large the error reaches 9 %. A similar error plot is shown in Fig. 8.14 for the CMOS implementation of the SUM building block at room temperature. We notice a very large error in the corners of the plot where the magnitude of the output is large. Here, the relative error can be as big as 70 %. This effect is due to the saturation of the differential voltage

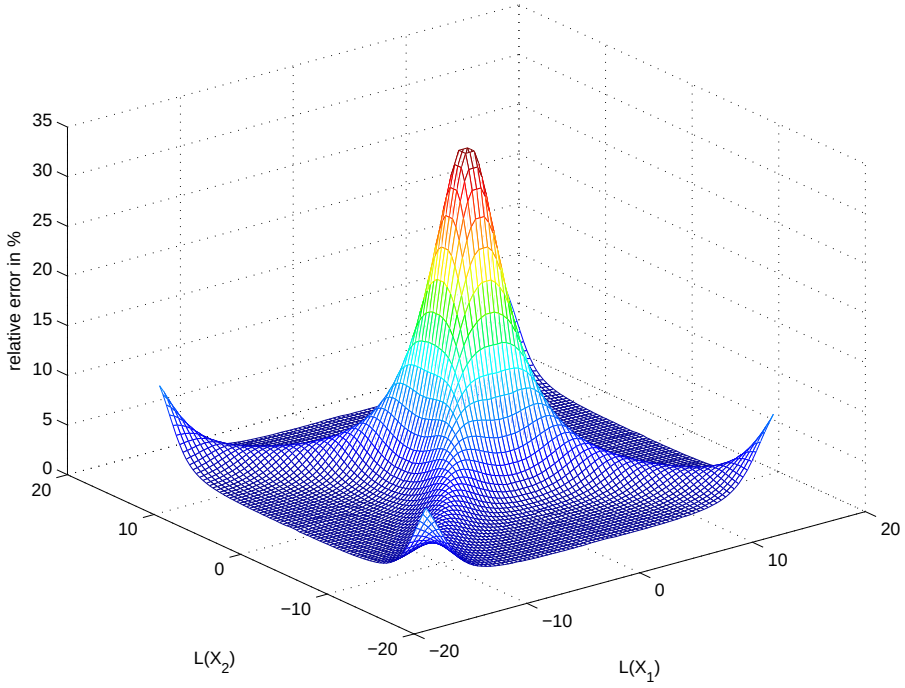



**Figure 8.13**: Relative error of the BPX building block at 27 °C.

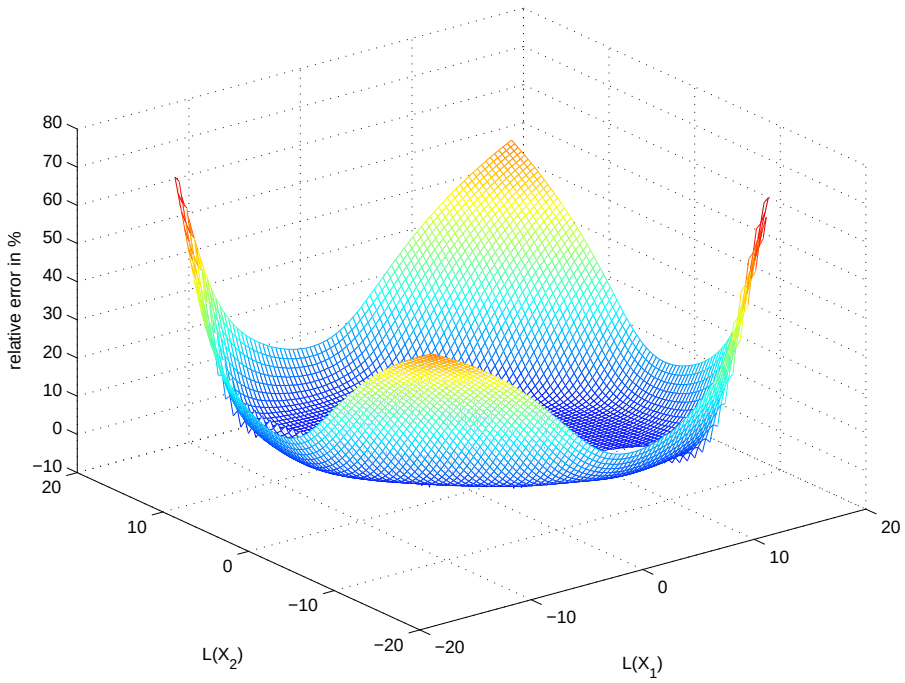



**Figure 8.14**: Relative error of the SUM building block at 27 °C.

at the output of this building block. The error introduced by the BPX and SUM building blocks increases at higher temperatures. Fig. 8.15 illustrates the relative error for the example of the BPX building block at 80 °C. Here, the relative error reaches 42 % for input values close to zero and when both inputs have a large magnitude it amounts up to 17 %.

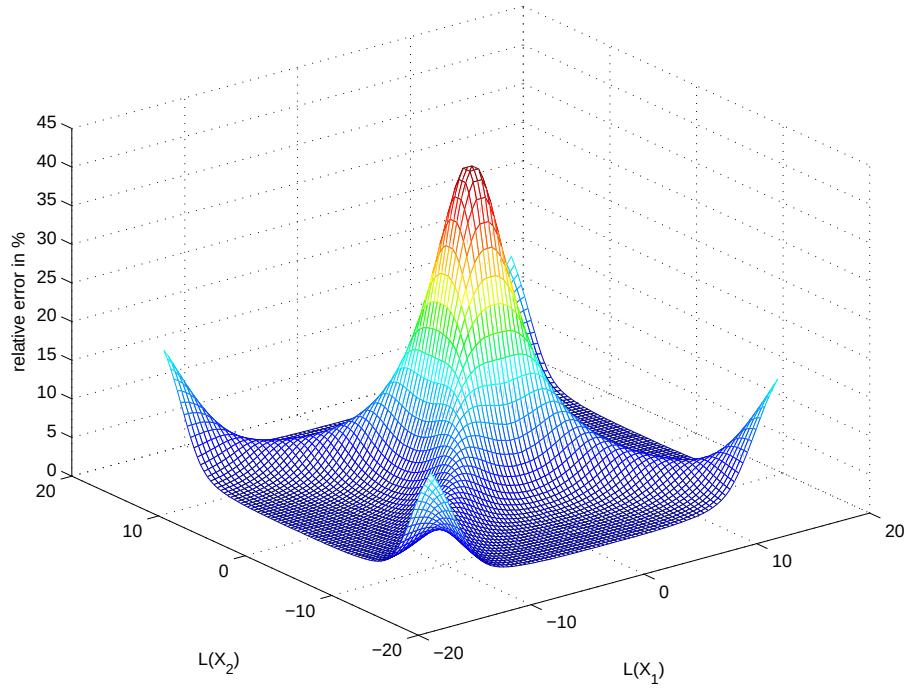



**Figure 8.15**: Relative error of the BPX building block at 80 °C.

Given the large error of the BPX and SUM building blocks it is in fact surprising to see that the overall LDPC decoder achieves a remarkable good BER performance close to optimal decoder performance, see Section 8.2.4. Clearly, this performance can only be achieved with trimming of the decoder.

### 8.3.2 Noise Effects

Digital circuits or analog radio frequency (RF) circuits may inject noise into the substrate. Substrate noise spectral components are typically found around the clock frequency of digital circuits or the characteristic frequencies of the circuit. Different noise isolation schemes are known which isolate noise more efficiently than simply increasing the distance to a noise source. This includes guard rings and deep Nwell (triple-well). These techniques are commonly used for system on chip (SOC) solutions and are thus offered in most CMOS processes. They can also be applied when it comes to the integration of an analog LDPC decoder into a SOC. Deep Nwell (DNW) achieves the best noise isolation because of an additional pn junction. Besides DNW, the transistors can also be protected by a guard ring or a combination of both.

The noise sensitivity of the analog LDPC decoder is generally reduced through the use of differential circuits throughout the decoder core. It is thus expected that the integration together with D/A and A/D converters as well as with digital signal processing blocks does not pose any problem for the performance of analog LDPC decoders.

### 8.3.3 Supply Voltage Variations

Small and slow variations of the supply voltage over time have only negligible impact on the differential output of a block and on overall decoder performance. In fact, the building blocks of the analog decoder core can handle supply voltages as low as 1.2 V at the expense of a small performance loss in terms of BER. AC components in the supply voltage in general change the single ended voltage levels within the circuits. The use of differential circuits throughout the analog decoder core reduces these effects similar to the case of noise introduced via the bulk. AC noise almost perfectly cancels out for the important case of differential signals near the decision threshold, i.e., a differential voltage of 0 V, although the single ended voltages carry the noise component. A fraction of the AC signal may become visible at large differential output voltages representing a high reliability of the output. Here, an AC component in the supply voltage with a magnitude of 10 mV and a frequency in the range of 10 kHz to 100 MHz[1] leads to an AC component in the maximum differential output voltage of a block with a magnitude of 2.5 mV. However, at this point the magnitude of the AC signal compared to the differential output signal is small and the decoder decision is not altered.

Noise components at the output of one block propagate to the input of a consecutive block. We therefore also investigated the case of noisy input signals (also with a magnitude of 10 mV) in combination with a noisy power supply and possible phase shifts between the noise signals. Any phase shift between the pairs of input wires or between the input and power supply leads to noise cancelation near the decision threshold. However, a phase shift between the noise on the two wires of a differential input leads to a noisy differential output signal over the full differential output voltage range. This effect can be avoided by matching the parasitics and thus the delay of the differential signal wires in the physical layout of the LDPC decoder.

The use of separate supply voltage sources for the digital and analog part of the chip may reduce or avoid the introduction of digital noise via the supply voltage.

### 8.3.4 Device Mismatch

One of the important questions in analog decoding is about the effect of transistor mismatch on the overall decoder performance. In the context of analog LDPC decoders it is important to match the transistors within one building block rather than between different building blocks. This implies that local mismatch (geometry dependent mismatch) is more important than global mismatch (spacing dependent mismatch). We therefore focus on local mismatch sources.

Mismatch can be divided into a mismatch of the threshold voltage $V_{th}$ between neighboring devices of equal size and so-called $\beta$-mismatch [LWMM98], [LHC86]. The drain current mismatch $\Delta I_D$ of the nMOS transistor in (6.6) can be expressed as

$$\frac{\Delta I_D}{I_D} = \frac{g_m}{I_D}\,\Delta V_{th} + \frac{\Delta\beta}{\beta}, \tag{8.6}$$

with $g_m = \partial I_D / \partial V_{GS}$ as the transconductance of the transistor and

$$\beta = \mu\,\mathrm{C_{ox}}\,\frac{\mathrm{W}}{\mathrm{L}}. \tag{8.7}$$

In terms of the variance we obtain [LHC86], [LWMM98]

$$\frac{\sigma^2(\Delta I_D)}{I_D^2} = 4\frac{\sigma^2(\Delta V_{th})}{(V_{GS}-V_{th})^2} + \frac{\sigma^2(\Delta\beta)}{\beta^2} \tag{8.8}$$

$$= \left(\frac{g_m}{I_D}\right)^2 \frac{A_{V_{th}}^2}{\mathrm{W}_{eff}\mathrm{L}_{eff}} + \frac{A_\beta^2}{\mathrm{W}_{eff}\mathrm{L}_{eff}}. \tag{8.9}$$

[1] A larger suppression is achieved outside this frequency range.

The constants $A_{V_{th}}$ and $A_\beta$ in (8.9) characterize the threshold voltage mismatch and $\beta$-mismatch, respectively. $W_{eff}$ and $L_{eff}$ refer to the effective physical dimensions of the transistor which are typically slightly smaller than the width and length specified in the design process. These parameters may be provided by the foundry as part of the design kit. Typically, the threshold voltage mismatch is the dominating effect so that the $\beta$-mismatch can be neglected. We therefore omit the second term on the right hand side of (8.9) in the following.

In case the parameter $A_{V_{th}}$ is not available for a given technology we can exploit the fact that $A_{V_{th}}$ increases roughly linearly with the gate oxide thickness $T_{ox}$ [DTS03]. For a gate oxide thickness of a TSMC technology with $T_{ox} = 4.1$ nm we obtain $A_{V_{th}} = 5$ mV·$\mu$m [DTS03]. We now investigate the drain current mismatch for transistors of size W$= 4 \times 1.5$ $\mu$m and W$= 8 \times 1.5$ $\mu$m, both with L$= 500$ nm. The effective transistor sizes are calculated based on the parameters provided in the design kit. Together with the simulation results for the transconductance $g_m$ we obtain a drain current mismatch which dependents on the drain current $I_D$ as shown in Fig. 8.16. Mismatch is reduced for a fixed transistor size when the drain current $I_D$ is increased. A similar effect is achieved when the effective area is increased for a given $I_D$. Note that doubling the area approximately decreases mismatch by a factor of $\sqrt{2}$. When we compare the estimated mismatch for a transistor with W$= 8 \times 1.5$ $\mu$m and L$= 500$ nm with the inherent error of the CMOS decoder building blocks in Section 8.3.1 we can conclude that the latter clearly dominates.

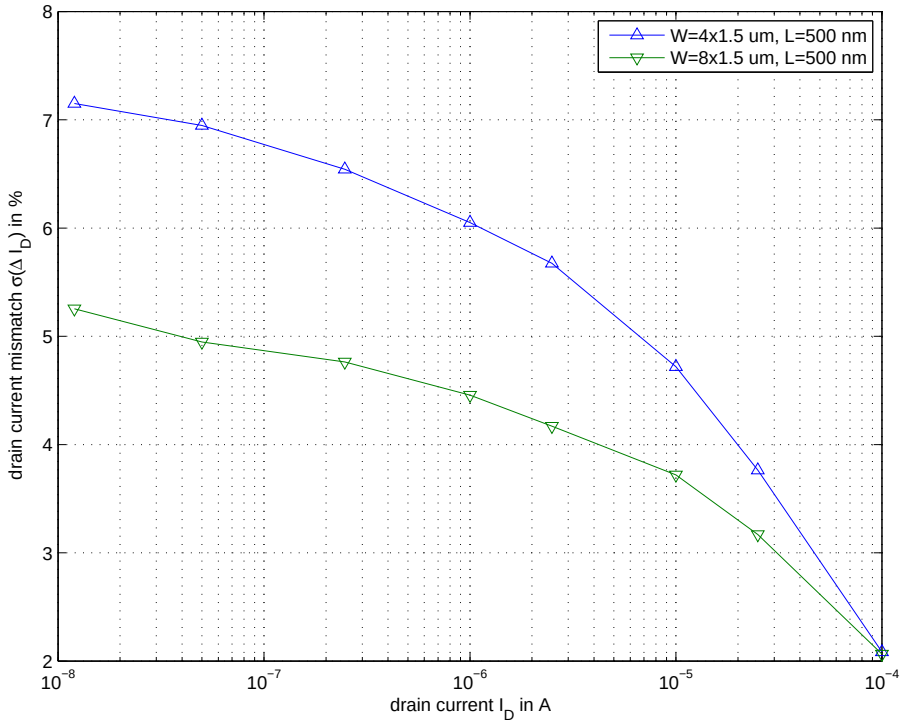



**Figure 8.16**: Drain current mismatch for nMOS transistors with different size.

The effect of device mismatch on the performance of analog decoders is also investigated in [LHL+99b], [Lus00], [AMNX02], [Dai02] and [Win04] without finding any significant degradation. Only a minor performance degradation is reported in [LL01] for a comparable device mismatch of 5 %. A table with very similar results as in Fig. 8.16 is presented in [Dai02] for a 0.5 $\mu$m CMOS technology. Dai also concluded that the error due to the quadric behavior of the devices (compared to the exponential behavior of bipolar transistors) is much larger than the error due to mismatch.

### 8.3.5 Process Variance

There are process variations across multiple wafers and fabrication cycles. A typical effect is, for example, the variation of the threshold voltage. The trimming technique in Section 8.2.4 also facilitates post-fabrication performance optimization. We demonstrated that it is possible to optimize the performance of an analog LDPC decoder over the full SNR and temperature range. It is therefore expected that process variations and changes of the threshold voltage can also be compensated.

### 8.3.6 Shrinking Device Sizes

Scaling of CMOS technology leads to improvements in device speed, matching and minimum noise figure. The disadvantage of scaling include reduced linearity and increased $1/f$ noise. Also, lowered nominal supply voltages go along with scaling. This usually poses a problem for analog circuit design. Our current circuit design can cope with supply voltages of 1.2 V and less at the expense of a small performance degradation. We can adjust to even lower supply voltages by changing the circuit architecture. This, however, does not necessarily lead to a similar clear advantage in terms of power consumption as in digital circuits. Low voltage architectures in the analog domain are in general slower than their high-voltage counterparts. From an analog point of view it is therefore desirable to benefit from shrinking device sizes while maintaining higher supply voltages as for digital circuits.

With advances in packaging technology many companies make use of system in a package (SIP) solutions rather than system on chip (SOC). The key advantage of this approach is that both analog and digital components can be implemented using the best suited technology without compromise on the performance. Digital designs can then directly benefit from speed and cost advantages of shrinking device sizes in CMOS while analog parts may still be manufactured in a technology with higher supply voltage. Several chips, i.e., dies, can then be grouped together in a single package which is also commonly referred as chip. Another approach is the use of so-called flip-chip technology which also allows the integration of different dies into one package. Both, SIP and flip-chip technology may significantly decrease the time to market in case of changes in the system or in case new features need to be added. They are therefore attractive alternatives for the integration of analog signal processing into a system.

## 8.4 Speed Estimation

The speed of analog decoders is determined by various different parameters. This includes the technology, the circuit architecture, the type and the size of the transistors, the physical layout of the decoder, the bias currents, temperature and also the configuration of channel values at the input of the decoder. The structure of the decoder building blocks in Chapter 6 makes the decoder very fast around the decision threshold. This is because the bias current is divided almost equally among the transistors in one row of the building block. Here, the current can quickly be steered from one transistor into another. This behavior is in strong contrast to the situation when a building block needs to provide a very reliable output, i.e., a large differential output voltage. In this case, almost all the bias current flows through one transistor in each row of the building block while the currents through the other transistors need to be decreased to almost zero. This operation is extremely time consuming and typically requires up to several orders of magnitude longer than switching near the decision threshold. Our speed estimations for analog LDPC decoders are based on such a scenario. For this, we considered a sub-block of the overall decoder and applied an input pulse with a rise time of 1 ns at the input and then simulated the transient response with the circuit-level simulator. We then measured the required time so that the output settles to 98 % of the largest possible differential output voltage. The results of these circuit-level simulations are depicted in Fig. 8.17 as a function of the bias

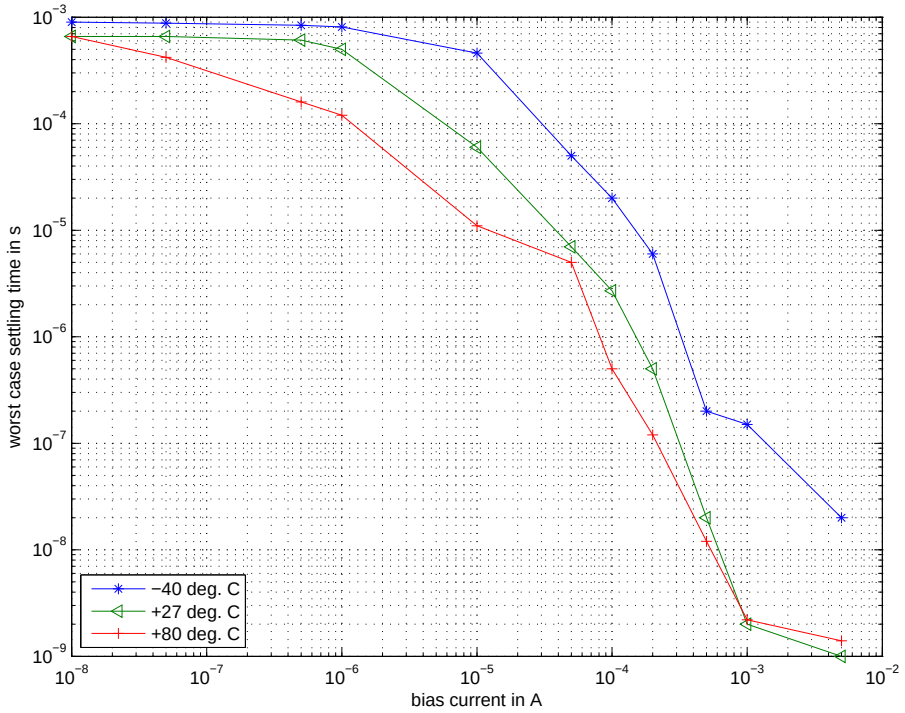



**Figure 8.17**: Worst case settling time of the analog decoding network at different temperatures (settling to 98 % of the largest possible magnitude at the output).

current $I_b$ of a building block. Temperatures of -40 °C, +27 °C and +80 °C are considered. This scenario is assumed to reflect the worst case settling behavior of the decoder. These results are derived for an analog LDPC decoder where the building blocks consist of nMOS devices with W=$8 \times 1.5$ $\mu$m and L=$500$ nm. The voltage-shifting output stages use the same nMOS devices with a bias current of $0.1$ $I_b$. Note that the settling time decreases when the temperature increases, i.e., the decoder operates faster at higher temperatures.

A speed estimation based on the worst case settling time in Fig. 8.17 would lead to a significant underestimation of decoder speed. This is because the settling time of outputs with a large magnitude, i.e., a high reliability, does not alter the decoder decisions and thus the BER performance. This reasoning is also supported by the fact that every summation building block of the decoder comprises a large error of up to 70 % for outputs with a large magnitude, see Section 8.3.1. From a decoding point of view the most critical range is around a zero differential output, i.e., the decision threshold, and this is where the decoder is up to several orders of magnitudes faster. In the following we assume that the appropriate hard decision can be provided at the decoder output within 1 % of this worst case settling time with no (or only a negligible) performance degradation. This assumption appreciates the fact that outputs with a large magnitude may not have settled. Fig. 8.18 and Fig. 8.19 depict the results of this speed estimation for the example of the rate $R = 1/2$ and rate $R = 5/6$ decoders, respectively. Again, the results are plotted as a function of the bias used for each building block of the decoder. As it can be expected from analog circuits, decoder speed increases when the bias currents are increased. It is interesting to see that both LDPC decoders potentially support data rates beyond 100 Gbps. Note that for a given bias current per block the rate $R = 5/6$ decoder reaches a higher throughput since $K = 540$ information bits are decoded in parallel instead of only $K = 324$ in the rate $R = 1/2$ decoder.

It is important to emphasize that there is not strict speed limitation on analog decoders. Analog decoders always degrade gracefully by producing a slightly increased BER whenever there

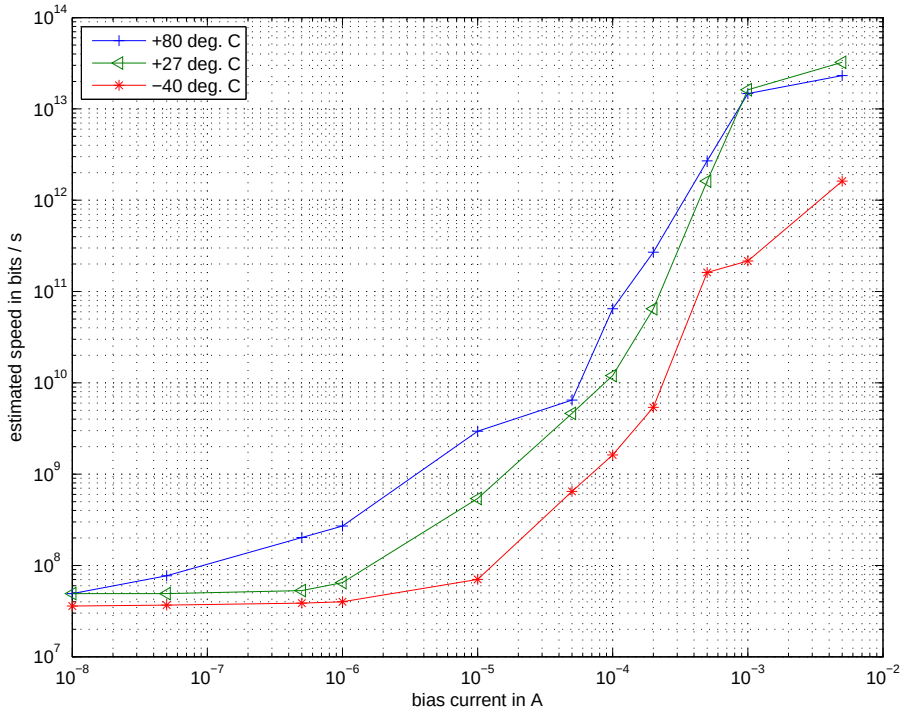



**Figure 8.18**: Estimated speed of the (648,324) analog LDPC decoder at different temperatures.

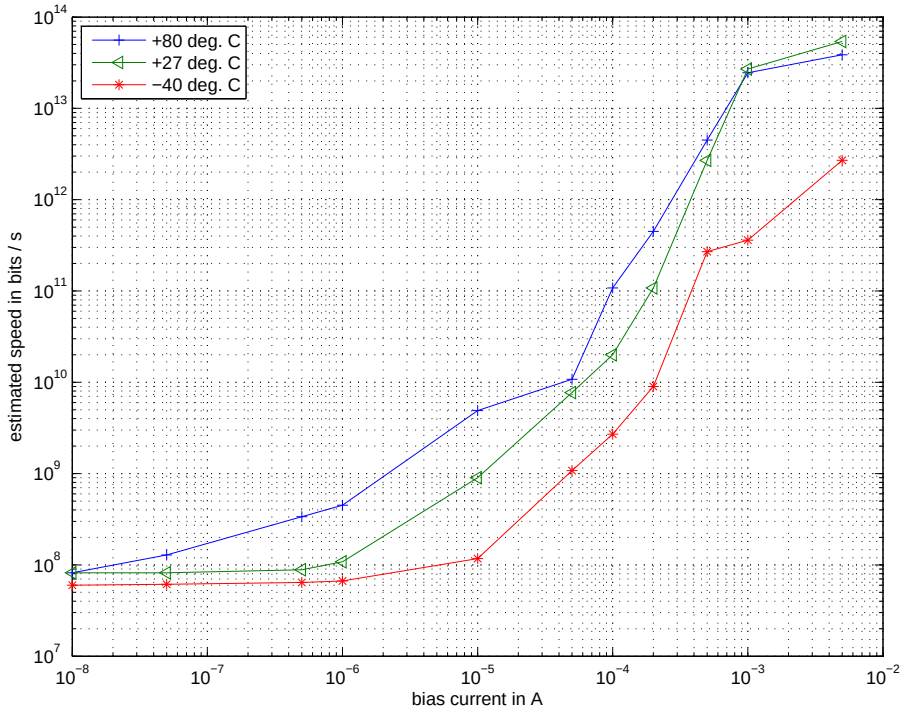



**Figure 8.19**: Estimated speed of the (648,540) analog LDPC decoder at different temperatures.

**Table 8.3**: Estimations for the two analog LDPC decoder cores.

| **decoder core** | | |
|---|---|---|
| LDPC decoder | $R = 1/2$ | $R = 5/6$ |
| $(N, K)$ | (648,324) | (648,540) |
| #BPX building blocks | 5184 | 6480 |
| #SUM building blocks | 5184 | |
| nMOS width W | $8 \times 1.5$ $\mu$m | |
| nMOS length L | 500 nm | |
| # of transistors | 135 k | 152 k |
| clock | - | |
| area<br>- guard ring + DNW per nMOS<br>- no guard ring, no DNW | <br>79 mm$^2$<br>13 mm$^2$ | <br>89 mm$^2$<br>15 mm$^2$ |
| speed @ power (27 °C)<br>$I_b = 50$ nA<br>$I_b = 1$ $\mu$A<br>$I_b = 10$ $\mu$A | <br>50 Mbps @ 1.1 mW<br>65 Mbps @ 22.4 mW<br>540 Mbps @ 224 mW | <br>80 Mbps @ 1.3 mW<br>110 Mbps @ 25.2 mW<br>900 Mbps @ 252 mW |
| energy per info bit (27 °C)<br>$I_b = 50$ nA<br>$I_b = 1$ $\mu$A<br>$I_b = 10$ $\mu$A | <br>0.022 nJ/bit<br>0.345 nJ/bit<br>0.415 nJ/bit | <br>0.016 nJ/bit<br>0.229 nJ/bit<br>0.280 nJ/bit |

is not enough time for some very time-demanding decoding situations. The BER performance is thus essentially a function of the settling time. The allowed settling time of an analog LDPC decoder is thus comparable to the number of iterations in a digital decoder implementation.

## 8.5 Estimated Performance Characteristics

In this section we estimate the performance characteristics of the overall LDPC decoder including input interface, D/A conversion, A/D conversion and output register. All estimations are based on the 0.18 $\mu$m CMOS technology offered from TSMC with a supply voltage of 1.8 V. The performance characteristics include transistor or gate[2] count, area and power consumption. The power estimations for the input and output interfaces including D/A conversion are based on a TSMC specification for the given technology. With a power supply of 1.8 V the maximum power consumption is 30 nW/MHz/gate. This allows us to estimate the power consumption based on the number of gates and the frequency of the corresponding clock signal.

### 8.5.1 Decoder Core

Our estimations for the two analog LDPC decoder cores are summarized in Table 8.3. The decoder core for the rate $R = 5/6$ code requires a larger number of BPX blocks and the same number of SUM building blocks as the decoder core for the $R = 1/2$ code. A transistor size of W$= 8 \times 1.5$ $\mu$m and L$= 500$ nm is assumed for the building blocks. The area estimation for the analog LDPC decoder core is based on the individual transistor sizes, the required routing within the building blocks, the interconnects between the building blocks and power routing. There are two options for the area estimation in Table 8.3. One is based on the assumption that each transistor is protected by guard ring and DNW and the other is without this shielding techniques. Note that the latter leads to a significant area reduction. A large fraction of this area

[2]One gate consists of two transistors.

reduction may be captured by shielding the building blocks rather than each individual transistor. The speed of the analog LDPC decoder is estimated together with the associated power consumption based on the results in Fig. 8.18 and Fig. 8.19 (room temperature).

### 8.5.2 Input Interface

The input interface consists of serial input registers, parallel data latches and bit switches for D/A conversion as outlined in Section 8.1.2. Our following estimations are based on a quantization of the decoder input with three and four bit. The results for the individual blocks of the input interface in Fig. 8.5 are summarized in Table 8.4. It is surprising to see that almost all the power is consumed by the input shift register. The results of this power estimations are comparable with the power consumption reported for the shift register in [WCC+05]. Note that a similar shift register is also required at the input of a digital LDPC decoder. Also note that the estimated power consumption of the D/A conversion is roughly three orders of magnitude lower than the power consumption of the input register.

**Table 8.4**: Estimations for the input interface.

| **input interface** | | | | |
|---|---|---|---|---|
| LDPC decoder | $R = 1/2$ | | $R = 5/6$ | |
| input resolution | 3 bit | 4 bit | 3 bit | 4 bit |
| **input register** | | | | |
| # of gates | 17500 | 23300 | 17500 | 23300 |
| clock | 480 MHz | | 720 MHz | |
| area | 0.26 mm$^2$ | 0.35 mm$^2$ | 0.26 mm$^2$ | 0.35 mm$^2$ |
| power | 252 mW | 336 mW | 378 mW | 504 mW |
| **latch** | | | | |
| # of gates | 9700 | 13000 | 9700 | 13000 |
| clock | 480 MHz/648 | | 720 MHz/648 | |
| area | 0.15 mm$^2$ | 0.20 mm$^2$ | 0.15 mm$^2$ | 0.20 mm$^2$ |
| power | 0.2 mW | 0.3 mW | 0.3 mW | 0.4 mW |
| **D/A** | | | | |
| # of gates | 9100 | 19400 | 9100 | 19400 |
| clock | 480 MHz/648 | | 720 MHz/648 | |
| area | 0.14 mm$^2$ | 0.30 mm$^2$ | 0.14 mm$^2$ | 0.30 mm$^2$ |
| power | 0.2 mW | 0.4 mW | 0.3 mW | 0.6 mW |
| **total # of gates** | **36300** | **55700** | **36300** | **55700** |
| **total area** | **0.55 mm$^2$** | **0.85 mm$^2$** | **0.55 mm$^2$** | **0.85 mm$^2$** |
| **total power** | **252.4 mW** | **336.7 mW** | **378.6 mW** | **505 mW** |

### 8.5.3 Output Interface

The output interface consists of A/D converters, parallel data latches and a serial output register as described in Section 8.1.3. A/D conversion is achieved by using $K$ comparators at the output of the analog LDPC decoder core. These comparators detect the sign of the differential output voltage and make a hard decision on the $K$ decoded information bits ($K = 324$ for $R = 1/2$ and $K = 540$ for $R = 5/6$). These bits are then stored in parallel data latches and passed on to the serial output register. The results for the individual blocks of the output interface in Fig. 8.6 are summarized in Table 8.5. Note that the power consumption of the A/D conversion and the data latches is significantly lower than the power consumption of the shift register at the output.

Note that a similar shift register is also required at the output of a digital LDPC decoder.

**Table 8.5**: Estimations for the output interface.

| **output interface** | | |
|---|---|---|
| LDPC decoder | $R = 1/2$ | $R = 5/6$ |
| **A/D** | | |
| # of gates | 810 | 1350 |
| clock | 240 MHz/324 | 600 MHz/540 |
| area | 0.012 mm$^2$ | 0.02 mm$^2$ |
| power | 0.02 mW | 0.03 mW |
| **latch** | | |
| # of gates | 1620 | 2700 |
| clock | 240 MHz/324 | 600 MHz/540 |
| area | 0.03 mm$^2$ | 0.04 mm$^2$ |
| power | 0.04 mW | 0.06 mW |
| **output register** | | |
| # of gates | 6150 | 10300 |
| clock | 240 MHz | 600 MHz |
| area | 0.09 mm$^2$ | 0.15 mm$^2$ |
| power | 44 mW | 111 mW |
| **total # of gates** | **8580** | **14350** |
| **total area** | **0.13 mm$^2$** | **0.21 mm$^2$** |
| **total power** | **44.06 mW** | **111.09 mW** |

### 8.5.4 Overall Analog LDPC Decoder

The performance characteristics of the analog LDPC decoder including the digital input and output interface are summarized in Table 8.6. The area of the overall analog LDPC decoder is clearly dominated by the area of the decoder core. In case of a guard ring and DNW for each nMOS transistor in the building blocks the cores of the $R = 1/2$ and $R = 5/6$ decoder occupy 79 mm$^2$ and 89 mm$^2$, respectively. Without these shielding techniques the area reduces to 13 mm$^2$ and 15 mm$^2$. This significant area reduction strongly motivates the use of more area-efficient shielding techniques, e.g., shielding of the decoder core or shielding the individual building blocks. With a bias current of $I_b = 10$ $\mu$A per building block the analog decoder core easily supports the required data rates of 240 Mbps ($R = 1/2$) and 600 Mbps ($R = 5/6$). We notice that most of the power is consumed by the digital shift registers in the input and output interfaces. This is mainly due to the shift register at the input which needs to process input samples with a resolution of three or four bits. The clock signals are here 480 MHz and 720 MHz. The shift register at the output is clocked with 240 MHz and 600 MHz since only the hard decisions of the decoder are carried. In case the input samples are represented with four bits the decoder core consumes approximately only one third of the overall power. Given the large number of up to 152 k analog transistors in the decoder core the LDPC decoder may only be manufactured in CMOS technology. BiCMOS or SiGe implementations do not appear to be feasible due to yield problems in the manufacturing.

**Table 8.6**: Summary of the performance characteristics of the overall analog LDPC decoder.

| **overall decoder** | | | | |
|---|---|---|---|---|
| LDPC decoder | $R = 1/2$ | | $R = 5/6$ | |
| $(N, K)$ | (648,324) | | (648,540) | |
| technology | 0.18 $\mu$m CMOS | | | |
| supply voltage | 1.8 V | | | |
| **decoder core** | | | | |
| # of transistors | 135 k | | 152 k | |
| area<br>- guard ring + DNW per nMOS<br>- no guard ring, no DNW | <br>79 mm$^2$<br>13 mm$^2$ | | <br>89 mm$^2$<br>15 mm$^2$ | |
| speed @ power (27 °C)<br>$I_b = 50$ nA<br>$I_b = 1$ $\mu$A<br>$I_b = 10$ $\mu$A | <br>50 Mbps @ 1.1 mW<br>65 Mbps @ 22.4 mW<br>**540 Mbps @ 224 mW** | | <br>80 Mbps @ 1.3 mW<br>110 Mbps @ 25.2 mW<br>**900 Mbps @ 252 mW** | |
| **input interface** | | | | |
| input resolution | **3 bit** | **4 bit** | **3 bit** | **4 bit** |
| # of gates | 36300 | 55700 | 36300 | 55700 |
| area | 0.55 mm$^2$ | 0.85 mm$^2$ | 0.55 mm$^2$ | 0.85 mm$^2$ |
| power | 252.4 mW | 336.7 mW | 378.6 mW | 505 mW |
| **output interface** | | | | |
| # of gates | 8580 | | 14350 | |
| area | 0.13 mm$^2$ | | 0.21 mm$^2$ | |
| power | 44.1 mW | | 111.1 mW | |
| **total # of transistors (analog)** | **135 k** | | **152 k** | |
| **total # of gates (digital)** | **44880** | **64280** | **50650** | **70050** |
| **total area**<br>**- guard ring + DNW per nMOS**<br>**- no guard ring, no DNW** | <br>**79.68 mm$^2$**<br>**13.68 mm$^2$** | <br>**79.98 mm$^2$**<br>**13.98 mm$^2$** | <br>**89.76 mm$^2$**<br>**15.76 mm$^2$** | <br>**90.06 mm$^2$**<br>**16.06 mm$^2$** |
| **total power** | **520.5 mW** | **604.8 mW** | **741.7 mW** | **868.1 mW** |

# 9

# *Summary and Outlook*

In this thesis we investigated a new type of analog signal processing in the FEC decoder of a digital communication system. The use of analog signal processors promises higher operating speed, lower power consumption and/or smaller footprint compared to conventional digital decoder implementations. Analog decoders are therefore well suited for complex, and thus area-demanding, coding schemes in high-speed communication systems as well as applications demanding ultra-low power consumption.

The major topics of this thesis can be summarized as follows:

▷ **Representation of Analog Decoding Networks**

Different decoding algorithms were presented as special instances of generalized message passing decoding based on code graphs. We found that normal graphs are well suited for the representation of analog decoding networks. Here, the nodes in the normal graph represent analog node processors while edges determine the message exchange between node processors. We transformed normal graphs in a way that each node only exhibits degree three. The directed view of such a normal graph then represented the analog decoding network as a block diagram.

▷ **Simulation of Analog Decoders**

We developed a comprehensive simulation environment which includes different time-continuous and time-discrete simulation models of analog decoders. This environment is linked with the circuit-level simulator so that high-level simulation models can easily be validated against circuit-level simulation results of the same decoder or an identical input configuration. Even for very simple decoding networks it was not feasible to run Monte Carlo simulations for the BER at circuit level. Many interesting coding schemes even prohibited circuit-level simulations of a fully parallel decoder architecture. We experienced that even high-level simulation models working in continuous time limit the block length of the code to a few hundred bits. We therefore developed faster time-discrete simulation models which capture the dynamics of analog decoders with sufficient accuracy. These simulation models were verified against other time-continuous simulation models for different analog decoding networks. We found a happy match between our time-continuous and time-discrete simulation models for all considered block codes, convolutional codes and turbo codes. We also investigated the effect of quantization on the performance of

analog decoders. The reason for this was twofold. First, quantization occurs whenever analog decoders utilize a digital input interface in order to be compatible with other digital components in the receiver. In this case, the complexity and power consumption of the D/A converters directly depends on the number of quantization levels. The second reason was that a sensitivity to quantization would also suggest a sensitivity to other impairments like discharge of analog memories, noise and device mismatch. We demonstrated that a quantization of the input values with three to four bits only marginally degrades decoder performance for both LDPC decoders and turbo decoders. We therefore concluded that the requirements for D/A conversion and analog storage are rather low and that a relatively large quantization error can be tolerated. This also indicated a robustness against other impairments.

▷ **Possible Equivalence between Analog and Digital Decoding**

Analog and digital decoders based on cycle-free code graphs were found to provide identical outputs. We also verified this for the case of tailbiting convolutional codes where the code graph forms a single loop. In these cases both analog and digital decoders are based on the same code graph. This is different for code graphs with loops. Digital decoders then iterate between the different component decoders while analog decoders operate on the overall code graph at the same time. In cases where the analog decoder is based on a transformed normal graph the two decoders are clearly not equivalent in a strict sense, but we found no evidence that node processors with degree three or the time-continuous operation impact BER performance. However, we presented two examples of simple block codes where further manipulations of the code graph improved the BER performance of both analog and digital decoders. Digital decoder implementations essentially work with quantized messages and rely on approximations in the node processors in order to lower the computational complexity. This causes a performance degradation compared to the ideal digital decoder we assumed throughout this thesis. From this point of view we can then argue that a carefully designed analog decoder is very likely to outperform a digital decoder implementation not only in terms of speed, power consumption and area, but also in terms of the BER.

▷ **Decoder Architectures**

We investigated fully parallel as well as sequential decoder architectures. Fully parallel decoders are based on the overall code graph so that all bits of a code word are decoded in parallel. This introduces the maximum amount of parallelism into the decoding network and thus achieves the highest throughput. The disadvantage of this approach is that the complexity of the decoder increases linearly with the block length. For many interesting applications this leads to an unfavorable tradeoff between area, power consumption and speed. Furthermore, the block length and the interleaver of the analog decoder are confined by the wiring. We therefore proposed a novel turbo decoder architecture which provides more flexibility in the design process so that given speed requirements can be matched. This architecture relies on a sequential processing of the component codes with a smaller analog decoder. Internal messages are then stored on memory elements which may simply be realized with a capacitor in the analog domain or based on digital memory elements. The latter necessitates additional internal A/D and D/A converters. The key advantage of this approach is that various different block lengths and interleaver structures can be realized at the expense of some additional control hardware for addressing the storage elements. A fully parallel analog turbo decoder for UMTS was found to require up to 5.9 million transistors. Our sequential architecture reduced the transistor count in the analog core to 16.6 k, i.e., by a factor of 355. We concluded that sequential architectures are essential whenever different block lengths and interleavers need to be supported, or, when the block length of the code exceeds a few hundred bits.

▷ **Circuit Design**

The happy match between analog transistor circuits and the operations in a decoder rests on the exponential characteristic of bipolar transistors. A similar characteristic may also be achieved with CMOS devices operating below threshold. It is characteristic of our circuit design that we solely use npn transistors or nMOS devices in the high-speed signal path thus avoiding slower pnp or pMOS devices. We presented a library of analog transistor circuits which is suited for the realization of arbitrary analog decoders. It is characteristic of these blocks that they operate very fast around the decision threshold while it is more time demanding to provide reliable outputs. The building blocks exhibit voltage inputs and voltage outputs so that no additional voltage to current conversion is required. The output of a block can thus easily be connected to a multiplicity of other blocks. In cases where the output needs to be adjusted to the input of another block we used output stages. Three different realizations of output stages were covered. We designed complete analog signal processors for different coding schemes and analyzed the complexity of different decoder architectures.

▷ **Implemented Decoder Chips in BiCMOS and SiGe**

As a proof of concept two prototypes of analog decoders were successfully fabricated and tested. The fist decoder chip was implemented in 0.25 $\mu$m BiCMOS. The measured BER demonstrated a close correlation to the simulated results of the ideal decoder over the full SNR range. The measured transient impulse response indicated that the chip supports an uncoded data rate of 160 Mbps while only dissipating 20 mW from a 3.3 V supply. The I/O bound chip area was 1.680 mm$^2$. To the best of our knowledge, this implementation represents the world's fully operational analog decoder chip. Our second decoder chip was fabricated in 0.25 $\mu$m SiGe. Here, the measurement results indicated an increasing offset of the BER towards higher SNR values. This performance degradation was caused by only a few bit positions while other bit positions closely approached the performance of the ideal decoder. We concluded that the suboptimal results for some bits positions originated in the measurement setup of the decoder. Based on the switching between two different input configurations we estimated that the SiGe chip supports an uncoded data rate of 800 Mbps while dissipating 127 mW from a 3.3 V supply. The I/O bound chip area of 1.638 mm$^2$ was almost identical to the first chip. Our second decoder chip is believed to be the fastest analog decoder chip to date. The estimated energy per information bit of the BiCMOS and SiGe decoder chip was 0.125 nJ/b and 0.159 nJ/b, respectively.

▷ **Case Study of Analog LDPC Decoders in CMOS**

We investigated analog LDPC decoders in 0.18 $\mu$m CMOS for a commercial application in the emerging wireless LAN standard IEEE 802.11n with data rates of 240 Mbps and 600 Mbps. We found that the BER performance highly depends on the physical dimensions of the transistors and the temperature. The decoders thus required a very careful parameter optimization which has been achieved with a new approach based on EXIT chart analysis. After optimization the BER performance of the reference decoder was closely approached over the full SNR and temperature range. This was despite the fact that individual node processors still introduced an error of up to 70 per cent. Low precision node processors thus achieved overall accuracy at decoder level. We addressed layout issues and covered impairments like noise, supply voltage variations, device mismatch and process variations. We concluded that all these impairments are small compared to the error introduced by the node processors. The non-exponential characteristic of CMOS devices was identified to be the dominating source of performance degradation. We estimated the speed of these analog LDPC decoders for different temperatures and found that speed

nicely scales with power consumption so that data rates of hundreds of Gbps could potentially be achieved. The decoders were analyzed together with digital input and output interfaces. We found that neither D/A nor A/D conversion is critical in terms of area or power consumption. The area was clearly dominated by the analog core of the decoder. With a shielding of every analog transistor we estimated an area of 80 $mm^2$ and 90 $mm^2$ for the overall decoders. Without this shielding the area reduced to 14 $mm^2$ and 16 $mm^2$, respectively. It became apparent that handling the incoming digital bit stream at these high data rates consumed 1.5 to 2 times the power of the analog core. The total power consumption was 605 mW and 868 mW at data rates of 240 Mbps and 600 Mbps, respectively. The power consumption of the analog decoder core alone was 224 mW and 252 mW. The transistor count of the analog cores was 135 k and 152 k while the interfaces contributed up to 70 k additional digital gates.

Based on the outcomes of this thesis we suggest further work in the following areas:

- We found that certain manipulations of the code graphs improved the BER performance of both analog and digital decoders. This effect could be investigated further.
- We noticed that the speed of analog node processors is very fast around the decision threshold while the settling to large, i.e., reliable, outputs is very time-demanding. This effect has not been considered in our system-level simulations. The simulation models for analog decoders could thus be further refined so that this behavior is reflected appropriately.
- We presented a sequential decoder architecture for turbo codes which can be considered as a major step towards more flexibility and scalability of analog decoders. A similar decoder architecture should also be investigated for LDPC codes so that different coding schemes can be supported with a single analog decoder core.
- Shielding of each individual transistor appears to be too conservative. A similar protection may be achieved with more area-efficient shielding techniques, e.g., at a block level. This would facilitate a dramatic area reduction of the analog decoder as already outlined in the above.
- The performance of analog decoders including the performance estimations of our work should be further validated through further prototype fabrications and measurement results.

We conclude this work with the notice that analog signal processing has many applications beyond coding. These include the equalization of ISI channels [HOMM99], [HMO00a], multi-user detection, multiple-input multiple-output (MIMO) detection [SGPMM06] and fast Fourier transform (FFT).

# A

# *Symbols and Notation*

## Symbols and Notation with First Occurrence:

| | |
|---|---|
| $a_i$ | realization of symbol variable $A_i$, 20 |
| $\boldsymbol{a}_{\mid I_{\mathcal{A}}(j)}$ | projection of symbol realization $\boldsymbol{a}$ onto index set $I_{\mathcal{A}}(j)$, 21 |
| $A_i$ | symbol variable, 20 |
| $A_{V_{th}}$ | constant characterizing threshold voltage mismatch, 159 |
| $A_\beta$ | constant characterizing beta mismatch, 159 |
| $\boldsymbol{A}$ | state matrix, 13 |
| $\mathcal{A}$ | symbol configuration space of symbol variables $A_i$, $i \in I_{\mathcal{A}}$, 21 |
| $\mathcal{A}_i$ | vector space over a finite field, 20 |
| $\boldsymbol{B}$ | input matrix, 13 |
| $\mathfrak{B}$ | full behavior, 24 |
| $\mathfrak{B}_{\mid \mathcal{X}}$ | projection of full behavior onto symbol variables, 24 |
| $c_i$ | code bit at position $i$, 5 |
| $c_H$ | number of ones in parity-check matrix, 126 |
| $\boldsymbol{c}$ | code word or code sequence, 8 |
| $\hat{\boldsymbol{c}}$ | estimated code word, 36 |
| $\boldsymbol{c}_a$ | code word $a$, 130 |
| $\boldsymbol{c}_b$ | code word $b$, 130 |
| $\boldsymbol{c}_k$ | $n_0$-tuple of code bits at position $k$, 11 |
| $C$ | channel capacity, 18 |
| $\mathrm{C_{ox}}$ | oxide capacitance, 100 |
| $\boldsymbol{C}$ | output matrix, 13 |
| $\mathcal{C}$ | code, 8 |
| $\mathcal{C}_j$ | local code, $j \in I_{\mathcal{C}}$, 21 |
| $\mathcal{C}^\perp$ | dual code, 9 |

$d_{c,j}$ number of ones in row $j$ of a matrix, 10
$d_{v,i}$ number of ones in column $i$ of a matrix, 10
$d_H$ Hamming distance, 8
$d_{min}$ minimum Hamming distance, 8
$D$ number of trellis sections decoded within one decoding window, 47
$\boldsymbol{D}$ transition matrix, 13
$E_b$ energy per information bit, 7
$E_S$ energy per transmitted symbol, 6
$f(D)$ feedforward polynomial of a convolutional encoder, 12
$g_m$ transconductance, 158
$g(D)$ rational transfer function of a convolutional encoder, 12
$\boldsymbol{G}$ generator matrix of a code, 9
$\boldsymbol{G}_l$ sub-matrix of $\boldsymbol{G}$, 11
$\boldsymbol{G}_{sys}$ systematic generator matrix, 9
$\boldsymbol{G}(D)$ polynomial generator matrix, 12
$\boldsymbol{G}_{sys}(D)$ systematic polynomial generator matrix, 13
$h$ step size of time-discrete simulation model, 61
$H(\omega)$ transfer function, 60
$\boldsymbol{H}$ parity-check matrix of a code, 9
$i(t)$ current signal, 60
$I$ current, 99
$I(X;Y)$ mutual information, 17
$I_j$ $j$-th current, 108
$I_A$ mutual information at the a priori decoder input, 53
$I_{A1}$ mutual information at the a priori input of the 1st component decoder, 53
$I_{A2}$ mutual information at the a priori input of the 2nd component decoder, 53
$I_b$ bias current, 97
$I_B$ base current, 98
$I_C$ collector current, 98
$I_D$ drain current, 99
$I_E$ mutual information at the extrinsic decoder output, 53
$I_{E1}$ mutual information at the extrinsic output of the 1st component decoder, 53
$I_{E2}$ mutual information at the extrinsic output of the 2nd component decoder, 53
$I_S$ transport saturation current 98
$I_{\mathcal{A}}$ index set of symbol variables $A_i$, $i \in I_{\mathcal{A}}$, 21
$I_{\mathcal{A}}(j)$ index of a subset of symbol variables $A_i$, $i \in I_{\mathcal{A}}(j)$, 21
$I_{\mathcal{C}}$ index set of local codes $\mathcal{C}_j$, $j \in I_{\mathcal{C}}$, 21
$I_{\mathcal{S}}$ index set of state variables $S_l$, $l \in I_{\mathcal{S}}$, 23
$I_{\mathcal{S}}(j)$ index of a subset of state variables $S_l$, $l \in I_{\mathcal{S}}(j)$, 23
$I_{\mathcal{U}}$ index set of information bits, 36
$I_{\mathcal{X}}$ index set of symbol variables $X_i$, $i \in I_{\mathcal{X}}$, 21
$I_{\mathcal{X}}(j)$ index of a subset of symbol variables $X_i$, $i \in I_{\mathcal{X}}(j)$, 21
$\boldsymbol{I}_K$ $K \times K$ identity matrix, 9
$\boldsymbol{I}_m$ $m \times m$ identity matrix, 15
$\boldsymbol{I}_N$ $N \times N$ identity matrix, 61
$\boldsymbol{I}_{N-K}$ $(N-K) \times (N-K)$ identity matrix, 9
$\Delta I$ differential current, 97
$\Delta I_C$ difference between two collector currents, 101
$\Delta I_D$ drain current mismatch, 158
$k$ Boltzmann's constant, 98
$k_0$ information bits per trellis section, 11
$K$ number of information bits per code word, 7

$l_x(X)$ log-likelihood of realization $x$ of random variable $X$, 39
$L$ stabilization length of a sliding window decoder, 47
L length of transistor, 100
$\mathrm{L}_{eff}$ effective length of transistor, 159
$L_c$ channel state information, 7
$L_c y$ L-value for bit $X$ obtained from the channel, 64
$L_c y_i$ L-value for bit $X_i$ obtained from the channel, 37
$L_c y_k^{(\nu)}$ L-value associated with $\nu$-th code bit in trellis section $k$, 45
$L_c \boldsymbol{y}$ vector of L-values obtained from the channel, 41
$L(X)$ L-value for bit $X$, 39
$L_{x,x'}(X)$ L-value for random variable $X$ based on outcomes $x$ and $x'$, 43
$L(\hat{X}_i)$ L-value for bit $X_i$ at decoder output, 37
$L(\hat{X}_k^{(\nu)})$ decoder output for the $\nu$-th code bit of trellis section $k$, 46
$L(\hat{U}_k)$ L-value for information bit $U_k$ at decoder output, 44
$L_a(X_i)$ a priori L-value for bit $X_i$ at decoder input, 37
$L_a(X_k^{(\nu)})$ a priori L-value for the $\nu$-th code bit of trellis section $k$, 46
$L_e(X)$ extrinsic L-value for bit $X$ at the decoder output 64
$L_e(X_i)$ extrinsic L-value for bit $X_i$ at decoder output, 37
$L_e(X_k^{(\nu)})$ extrinsic L-value for the $\nu$-th code bit of trellis section $k$, 46
$L_{e,1}(U_k)$ extrinsic L-value for information bit $U_k$ at the output of the 1st component decoder, 51
$L_{e,2}(U_k)$ extrinsic L-value for information bit $U_k$ at the output of the 2nd component decoder, 51
$L_{in,1}(U_k)$ L-value for information bit $U_k$ at the input of the 1st component decoder, 50
$L_{in,2}(U_k)$ L-value for information bit $U_k$ at the input of the 2nd component decoder, 51
$L_\alpha(\cdot)$ L-value of state variable $(\cdot)$ (forward direction), 64
$L_\beta(\cdot)$ L-value of state variable $(\cdot)$ (backward direction), 64
$\boldsymbol{L}_{dec}$ vector of L-values at decoder output, 129
$LR(X)$ likelihood ratio for bit $X$, 39
$m$ encoder memory, 12
$m_a$ mean of Gaussian distributed L-values at the a priori decoder input, 53
$m_{y'}$ mean of Gaussian distributed L-values obtained from the channel, 52
$n(t)$ sample function of additive noise process, 6
$n_i$ realization of a Gaussian distributed random variable, 6
$n_0$ code bits per trellis section, 11
$N$ number of code bits per code word, 7
$N_0$ one-sided power spectral density of additive noise process, 6
$p(\cdot)$ probability density function, 17
$p(\cdot|\cdot)$ conditional probability density function, 7
$P_b$ probability of a transmission error, 7
$P_w$ word error probability, 36
$P_X(x)$ probability of realization $x$ of random variable $X$, 39
$P(\cdot)$ probability, 36
$P(\cdot|\cdot)$ conditional probability, 36
$q$ charge of an electron, 98
$q(D)$ feedback polynomial of a convolutional encoder, 12
$Q$ number of quantization levels, 54
$r$ bit resolution, 146
$r(t)$ time-continuous received signal, 5
$R$ code rate, 7
$R_0$ computational cutoff rate, 54

$s$ realization of state variable $S$ at time $k$, 44
$s'$ realization of state variable $S'$ at time $k+1$, 44
$s(t)$ time-continuous transmit signal, 5
$\boldsymbol{s}(t)$ state vector, 60
$s_{c_i}(t)$ time-continuous transmit signal representing code bit $c_i$, 5
$\boldsymbol{s}_k$ encoder state at time $k$, 13
$\boldsymbol{s}_{|I_S(j)}$ projection of state realization $\boldsymbol{s}$ onto index set $I_S(j)$, 24
$\boldsymbol{s}_k^{[zi]}$ zero-input solution at time $k$, 14
$\boldsymbol{s}_k^{[zs]}$ zero-state solution at time $k$, 14
$S_l$ state variable at position $l$, $l \in I_S$, 23
$S$ state variable at time $k$, 44
$S'$ state variable at time $k+1$, 44
$\mathcal{S}$ state configuration space of state variables $S_l$, $l \in I_S$, 24
$t$ time, 5
$\Delta t$ time step, 61
$T$ absolute temperature in degrees Kelvin, 98
$T_{ox}$ gate oxide thickness, 159
$T_S$ symbol duration, 6
$T_\nu$ quantization thresholds, $\nu \in \{1, \ldots, Q-1\}$ 55
$u_k$ information bit at position $k$, 4
$\hat{u}_k$ estimation for information bit at position $k$, 5
$\boldsymbol{u}$ block/sequence of information bits, 8
$\boldsymbol{u}_a$ information word $a$, 130
$\boldsymbol{u}_b$ information word $b$, 130
$\boldsymbol{u}_k$ encoder input at time $k$, 14
$v(t)$ voltage signal, 60
$\boldsymbol{v}(t)$ voltage vector, 61
$V$ single-ended voltage, 99
$V_j$ $j$-th single-ended voltage, 100
$V_{i,j}$ differential voltage, 108
$V_{BE}$ base-emitter voltage, 98
$V_{DD}$ supply voltage, 102
$V_{DS}$ drain-source voltage, 99
$V_{GS}$ gate-source voltage, 99
$V_{ref}$ reference voltage, 130
$V_{th}$ threshold voltage, 99
$V_T$ thermal voltage, 97
$\Delta V$ differential voltage, 97
$\Delta V_{th}$ threshold voltage mismatch, 158
$\boldsymbol{V}_i$ voltage vector, 109
$\boldsymbol{V}_{L_c y}$ voltage vector representing $L_c y$, 115
$\boldsymbol{V}_{L_c y_k^{(i)}}$ voltage vector representing $L_c y_k^{(i)}$, 119
$\boldsymbol{V}_{L_e(\cdot)}$ voltage vector representing $L_e(\cdot)$, 116
$\boldsymbol{V}_{L_\alpha(\cdot)}$ voltage vector representing $L_\alpha(\cdot)$, 116
$\boldsymbol{V}_{L_\beta(\cdot)}$ voltage vector representing $L_\beta(\cdot)$, 116
$\boldsymbol{V}^*_{L_\beta(S')}$ voltage vector with offset representing $L_\beta(S')$, 116
$\boldsymbol{V}_{\boldsymbol{\alpha}_k}$ voltage vector representing $\boldsymbol{\alpha}_k$, 119
$\boldsymbol{V}_{\boldsymbol{\alpha}_{k+1}}$ voltage vector representing $\boldsymbol{\alpha}_{k+1}$, 120
$\boldsymbol{V}_{\boldsymbol{\beta}_k}$ voltage vector representing $\boldsymbol{\beta}_k$, 120
$\boldsymbol{V}_{\boldsymbol{\beta}_{k+1}}$ voltage vector representing $\boldsymbol{\beta}_{k+1}$, 119
$\boldsymbol{V}^*_{\boldsymbol{\gamma}_k}$ voltage vector representing $\boldsymbol{\gamma}_k$, 119

| | |
|---|---|
| $W$ | window size of sliding window decoder, 47 |
| $\mathrm{W}$ | width of transistor, 100 |
| $\mathrm{W}_{eff}$ | effective width of transistor, 159 |
| $W_T$ | trellis sections of a tailbiting convolutional code, 49 |
| $x$ | realization of symbol variable $X$, 64 |
| $x_i$ | realization of symbol variable $X_i$, 21 |
| $\hat{x}$ | estimation for bit $x$, 40 |
| $\boldsymbol{x}$ | vector of symbol realizations, 8 |
| $\boldsymbol{x}_{\mid I_{\mathcal{X}}(j)}$ | projection of symbol realizations $\boldsymbol{x}$ onto index set $I_{\mathcal{X}}(j)$, 24 |
| $X$ | symbol variable, 17 |
| $X_i$ | symbol variable at position $i$, $i \in I_{\mathcal{X}}$, 21 |
| $\mathcal{X}$ | symbol configuration space of symbol variables $X_i$, $i \in I_{\mathcal{X}}$, 8 |
| $\mathcal{X}_i$ | vector space over a finite field, 21 |
| $y$ | noisy received sample, 64 |
| $y_i$ | noisy received sample at position $i$, 7 |
| $\bar{y}^{(\nu)}$ | $\nu$-th quantization level of noisy received sample, $\nu \in \{1, \ldots, Q\}$, 54 |
| $\boldsymbol{y}$ | vector of received samples, 36 |
| $Y$ | value-continuous random variable, 17 |
| $z$ | expansion factor of LDPC code, 147 |
| $\alpha_F$ | base transport factor, 99 |
| $\alpha_k(s)$ | metric of forward recursion at time $k$, 45 |
| $\boldsymbol{\alpha}_k$ | row vector of forward metrics at time $k$, 46 |
| $\beta_F$ | forward current gain, 99 |
| $\beta_{k+1}(s')$ | metric of backward recursion at time $k+1$, 45 |
| $\boldsymbol{\beta}_k$ | column vector of backward metrics at time $k$, 46 |
| $\gamma_k(s, s')$ | transition metric from state $s$ to state $s'$ at time $k$, 45 |
| $\gamma_k^{(\nu)}(s, s')$ | transition metric from state $s$ to state $s'$ at time $k$ associated with $\nu$-th code bit, 45 |
| $\boldsymbol{\Gamma}_k$ | transition matrix at time $k$, 46 |
| $\Delta\beta$ | beta mismatch, 158 |
| $\lambda(X)$ | soft bit of binary random variable $X$, 40 |
| $\mu$ | surface mobility of electrons, 100 |
| $\nu$ | constraint length, 12 |
| $\sigma_a^2$ | variance of Gaussian distributed L-values at the a priori decoder input, 53 |
| $\sigma_n^2$ | variance of Gaussian noise, 6 |
| $\sigma_{y'}^2$ | variance of Gaussian distributed L-values received from the channel, 52 |
| $\tau$ | time constant, 60 |
| $\psi$ | log-likelihood offset, 97 |
| $\Psi$ | voltage offset, 108 |
| $\omega$ | angular frequency, 60 |

# B

# Abbreviations

## Abbreviations with First Occurrence:

| | |
|---|---|
| A/D | analog-to-digital, 54 |
| APP | a posteriori probability, 36 |
| ASIC | application specific integrated circuit, 57 |
| AWGN | additive white Gaussian noise, 6 |
| BCJR | Bahl Cocke Jelinek Raviv, 35 |
| BER | bit error rate, 8 |
| BiCMOS | bipolar CMOS, 3 |
| BPSK | binary phase shift keying, 5 |
| BPX | Boxplus, 145 |
| CMOS | complementary metal oxide semiconductor, 97 |
| CNP | check node processor, 118 |
| CSI | channel state information, 7 |
| DAQ | data acquisition, 139 |
| D/A | digital-to-analog, 54 |
| DMC | discrete memoryless channel, 54 |
| DNW | deep Nwell, 157 |
| DSP | digital signal processor, 57 |
| DVB | digital video broadcast, 8 |
| EDA | electronic design automation, 63 |
| EXIT | extrinsic information transfer, 52 |
| FEC | forward error correction, 1 |
| FFT | fast Fourier transform, 170 |
| FPGA | field programmable gate array, 57 |
| Gbps | gigabit per second, 142 |
| HSDPA | high-speed downlink packet access, 124 |
| I/O | input/output, 31 |
| ISI | inter-symbol interference, 48 |
| LAN | local area network, 3 |
| LDPC | low-density parity-check, 1 |

| | |
|---|---|
| MAP | maximum a posteriori, 36 |
| Mbps | megabit per second, 124 |
| MD | minimum distance, 36 |
| MIMO | multiple-input multiple-output, 170 |
| ML | maximum likelihood, 36 |
| MLF | micro leadframe, 142 |
| MSGM | minimum span generator matrix, 22 |
| MSPCM | minimum span parity-check matrix, 22 |
| PCB | printed circuit board, 58 |
| pdf | probability density function, 7 |
| pMUX | probability multiplexor, 108 |
| $\text{pMUX}^{-1}$ | inverse pMUX, 109 |
| RC | resistor-capacitor, 59 |
| RCPC | rate-compatible punctured convolutional codes, 14 |
| RF | radio frequency, 157 |
| SiGe | silicon germanium, 3 |
| SIP | system in a package, 160 |
| SNR | signal-to-noise ratio, 6 |
| SOC | system on chip, 157 |
| SPC | single parity-check, 9 |
| SUM | summation, 145 |
| SwinDec | (analog) sliding window decoding, 79 |
| TSMC | Taiwan semiconductor manufacturing company, 145 |
| UMTS | universal mobile telecommunications system, 15 |
| VNP | variable node processor, 118 |
| XOR | exclusive OR, 32 |

# *Bibliography*


[ABM+04] A. Graell i Amat, S. Benedetto, G. Montorsi, D. Vogrig, A. Neviani, and A. Gerosa. An analog turbo decoder for the UMTS standard. In *Proc. IEEE Int. Symposium on Information Theory*, page 296, Chicago, IL, USA, June/July 2004.

[ABM+05] A. Graell i Amat, S. Benedetto, G. Montorsi, D. Vogrig, A. Neviani, and A. Gerosa. An analog turbo decoder for the rate-1/3, 40 bit, UMTS turbo code. In *Proc. IEEE Int. Conference on Communications*, volume 1, pages 663–667, Seoul, Korea, May 2005.

[AF70] K. Abend and B. D. Fritchman. Statistical detection for communication channels with intersymbol interference. *Proceedings of the IEEE*, 58(5):779–785, May 1970.

[AG78] A. Acampora and R. Gilmore. Analog Viterbi decoding for high speed digital satellite channels. *IEEE Transactions on Communications*, 26(10):1463–1470, October 1978.

[AH98] J. B. Anderson and S. M. Hladik. Tailbiting MAP decoders. *IEEE Journal on Selected Areas in Communications*, 16(2):297–302, February 1998.

[ALJS04] M. Arzel, C. Lahuec, M. Jezequel, and F. Seguin. Analogue decoding of duo-binary codes. In *Proc. Int. Symposium on Information Theory and its Applications ISITA 2004*, Parma, Italy, October 2004.

[ALS+05] M. Arzel, C. Lahuec, F. Seguin, D. Gnaedig, and M. Jezequel. Analog slice turbo decoding. In *Proc. IEEE Int. Symposium on Circuits and Systems*, pages 332–335, Kobe, Japan, May 2005.

[ALSJ05] M. Arzel, C. Lahuec, F. Seguin, and M. Jezequel. Semi-iterative analogue turbo decoding. In *Proc. 4th Analog Decoding Workshop*, Rennes, France, June 2005.

[Ama04] A. Graell i Amat. *High-rate Convolutional Codes for High-speed Concatenated Codes Applications: Design and Efficient Decoding*. PhD thesis, Politecnico Di Torino, Turino, Italy, 2004.

[AMNX02] A. Graell i Amat, G. Montorsi, A. Neviani, and A. Xotta. An analog decoder for concatenated magnetic recording schemes. In *Proc. IEEE Int. Conference on Communications*, pages 1563–1568, New York, NY, USA, April/May 2002.

[Arz06] M. Arzel. *Semi-Iterative Analogue Turbo Decoding - An Application to DVB-RSC-Like Codes*. PhD thesis, École Nationale Supérieure des Télécommunications de Bretagne, Brest, France, 2006.

[ASLJ06] M. Arzel, F. Seguin, C. Lahuec, and M. Jezequel. Semi-iterative analog turbo decoding. In *Proc. IEEE Int. Symposium on Circuits and Systems*, Island of Kos, Greece, May 2006.

[BCJR74] 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.

[BDMP96] S. Benedetto, D. Divsalar, G. Montorsi, and F. Pollara. Soft-output decoding algorithms for continuous decoding of parallel concatenated convolutional codes. In *Proc. IEEE Int. Conference on Communications*, volume 1, pages 112–117, Dallas, TX, USA, June 1996.

[Ber00] J. Berkmann. *Iterative Decoding of Nonbinary Codes*. PhD thesis, Technische Universität München, Munich, Germany, 2000.

[BGT93] C. Berrou, A. Glavieux, and P. Thitimajshima. Near shannon-limit error-correcting coding and decoding: Turbo codes. In *Proc. IEEE Int. Conference on Communications*, volume 2, pages 1064–1070, Geneva, Switzerland, May 1993.

[CFV99] A. R. Calderbank, G. D. Forney, Jr., and A. Vardy. Minimal tail-biting trellises: The Golay code and more. *IEEE Transactions on Information Theory*, 45:1435–1455, July 1999.

[CHIW98] D. J. Costello, Jr., J. Hagenauer, H. Imai, and S. B. Wicker. Applications of error-control coding. *IEEE Transactions on Information Theory*, 44(6):2531–2560, October 1998.

[Dai02] J. Dai. *Design Methodology for Analog VLSI Implementations of Error Control Decoders*. PhD thesis, Univ. of Utah, 2002.

[Daw96] H. Dawid. *Algorithmen und Schaltungsarchitekturen zur Maximum a Posteriori Faltungsdecodierung*. PhD thesis, RWTH Aachen, Aachen, Germany, 1996.

[DCK05] A. Darabiha, A. C. Carusone, and F. R. Kschischang. Multi-Gbit/sec low density parity check decoders with reduced interconnect complexity. In *Proc. IEEE Int. Symposium on Circuits and Systems*, Kobe, Japan, May 2005.

[Dei91] A. S. Deif. *Advanced Matrix Theory for Scientists and Engineers*. Gordon and Breach, New York, 2nd edition, 1991.

[DGM93] H. Dawid, G. Gehnen, and H. Meyr. MAP channel decoding: Algorithm and VLSI architecture. In *Proc. Workshop on VLSI Signal Processing VI*, pages 141–149, Veldhoven, The Netherlands, October 1993.

[DJM98] D. Divsalar, H. Jin, and R. J. McEliece. Coding theorems for 'turbo-like' codes. In *Proc. Annual Allerton Conference on Communication, Control, and Computing*, pages 201–210, Allerton House, Monticello, IL, USA, September 1998.

[DT98] A. Demosthenous and J. Taylor. BiCMOS add-compare-select units for Viterbi decoders. In *Proc. IEEE Int. Symposium on Circuits and Systems*, pages 209–212, Monterey, CA, USA, May/June 1998.

[DTS03] C. H. Diaz, D. D. Tang, and J. Sun. CMOS technology for MS/RF SoC. *IEEE Transactions on Electronic Devices*, 50(3):557–566, March 2003.

[EPG94] J. Erfanian, S. Pasupathy, and G. Gulak. Reduced complexity symbol detectors with parallel structure for ISI channels. *IEEE Transactions on Communications*, 42(2/3/4):1661–1671, Feb./Mar./Apr. 1994.

[ETS00] ETSI - European Telecommunications Standards Institute. Universal mobile telecommunications system (UMTS): Multiplexing and channel coding (FDD). Technical Report 3GPP TS 125.212 ver. 3.4.0, September 2000.

[ETS03] ETSI - European Telecommunications Standards Institute. Digital video broadcasting (DVB); interaction channel for satellite distribution systems. Technical Report ETSI EN 301 790 ver. 1.3.1, March 2003.

[FKLW97] B. Frey, F. Kschischang, H.-A Loeliger, and N. Wiberg. Factor graphs and algorithms. In *Proc. Annual Allerton Conference on Communication, Control, and Computing*, pages 666–680, Allerton House, Monticello, IL, USA, Sept./Oct. 1997.

[FLL+04] M. Frey, H.-A. Loeliger, F. Lustenberger, P. Merkli, and P. Strebel. Measurements on an analog (8,4,4) Hamming code decoder chip. Technical report, Signal and Information Processing Laboratory, ETH Zurich, Switzerland, 2004.

[FLMS05] M. Frey, H.-A. Loeliger, P. Merkli, and P. Strebel. Two experimental analog decoders. In *Proc. IEEJ Int. Analog VLSI Workshop*, Bordeaux, France, October 2005.

[FM91] G. Fettweis and H. Meyr. High-speed parallel Viterbi decoding: algorithm and VLSI-architecture. *IEEE Transactions on Communications*, 29(5):46–55, May 1991.

[For66] G. D. Forney, Jr. *Concatenated Codes*. MIT Press, Cambridge, MA, USA, 1966.

[For01] G. D. Forney, Jr. Codes on graphs: normal realizations. *IEEE Transactions on Information Theory*, 47(2):520–548, February 2001.

[Fri95] B. Friedrichs. *Kanalcodierung - Grundlagen und Anwendungen in modernen Kommunikationssystemen*. Springer, Berlin, Germany, 1995.

[Gal62] R. G. Gallager. Low-density parity-check codes. *IRE Trans. Inform. Theory*, IT-8:21–28, January 1962.

[Gal63] R. G. Gallager. *Low-Density Parity-Check Codes*. MIT Press, Cambridge, MA, USA, 1963.

[Gau03] V. Gaudet. *Architecture and Implementation of Analog Iterative Decoders*. PhD thesis, Univ. of Toronto, Toronto, Canada, 2003.

[GG03a] V. Gaudet and G. Gulak. A 13.3Mbps 0.35$\mu$m CMOS analog turbo decoder IC with a configurable interleaver. *IEEE Journal of Solid-State Circuits*, 38(11):2010–2015, November 2003.

[GG03b] V. Gaudet and G. Gulak. A 13.3Mb/s 0.35$\mu$m CMOS analog turbo decoder IC with a configurable interleaver. In *Proc. IEEE Int. Solid-State Circuits Conference*, San Francisco, CA, USA, February 2003.

[GGG02] V. Gaudet, R. Gaudet, and G. Gulak. Programmable interleaver design for analog iterative decoders. *IEEE Transactions on Circuits and Systems—Part II: Analog and Digital Signal Processing*, 49(7):457–464, July 2002.

[Gil68] B. Gilbert. A precise four-quadrant multiplier with sub-nanosecond response. *IEEE Journal of Solid-State Circuits*, SC-3:1174–1178, December 1968.

[GM93] P. R. Gray and R. G. Meyer. *Analysis and Design of Analog Integrated Circuits*. John Wiley & Sons, Inc., New York, NY, USA, 3rd edition, 1993.

[Hag88] J. Hagenauer. Rate-compatible punctured convolutional codes (RCPC codes) and their applications. *IEEE Transactions on Communications*, 36(4):389–400, April 1988.

[Hag94] J. Hagenauer. Soft is better than hard. In R. E. Blahut, D. J. Costello, Jr., U. Maurer, and T. Mittelholzer, editors, *Communications and Cryptography - Two Sides of one Tapestry*, pages 155–171. Kluwer Academic Publishers, Dordrecht, The Netherlands, 1994.

[Hag97a] J. Hagenauer. The turbo principle - tutorial introduction and state of the art. In *Proc. Int. Symposium on Turbo Codes and Related Topics*, pages 1–11, Brest, France, September 1997.

[Hag97b] J. Hagenauer. Verfahren und Einrichtung zur analogen Detektion und Decodierung. German Patent DE 19725275, May 1997.

[Hag98] J. Hagenauer. Decoding of binary codes with analog networks. In *Proc. IEEE Information Theory Workshop*, pages 13–14, San Diego, CA, USA, February 1998.

[HB04] S. Hemati and A. H. Banihashemi. Dynamics and performance analysis of analog iterative decoding for low-density parity-check (LDPC) codes. In *Proc. IEEE Global Communications Conference*, pages 356–360, Dallas, TX, USA, November 2004.

[HB06] S. Hemati and A. H. Banihashemi. Dynamics and performance analysis of analog iterative decoding for low-density parity-check (LDPC) codes. *IEEE Transactions on Communications*, 54(1):61–70, January 2006.

[HBP05] S. Hemati, A. H. Banihashemi, and C. Plett. An 80-Mb/s 0.18-$\mu$m CMOS analog min-sum iterative decoder for a (32,8,10) LDPC code. In *Proc. IEEE Custom Integrated Circuits Conference*, pages 243–246, San Jose, CA, USA, September 2005.

[HBP06] S. Hemati, A. H. Banihashemi, and C. Plett. A 0.18-$\mu$m CMOS analog min-sum iterative decoder for a (32,8) low-density parity-check (LDPC) code. *IEEE Journal of Solid-State Circuits*, 41(11):2531–2540, November 2006.

[HC00] K. He and G. Cauwenberghs. Integrated 64-state parallel analog Viterbi decoder. In *Proc. IEEE Int. Symposium on Circuits and Systems*, Geneva, Switzerland, May 2000.

[HLL+99] M. Helfenstein, F. Lustenberger, H.-A. Loeliger, F. Tarköy, and G. S. Moschytz. High-speed interfaces for analog, iterative VLSI decoders. In *Proc. IEEE Int. Symposium on Circuits and Systems*, volume 2, pages 424–427, Orlando, FL, USA, May/June 1999.

[HMO00a] J. Hagenauer, M. Moerz, and E. Offer. Analog turbo networks in VLSI: The next step in turbo decoding and equalization. In *Proc. Int. Symposium on Turbo Codes and Related Topics*, pages 209–218, Brest, France, September 2000.

[HMO00b] J. Hagenauer, M. Moerz, and E. Offer. A circuit-based interpretation of analog MAP decoding with binary trellises. In *Proc. Int. ITG Conference on Source and Channel Coding*, pages 175–180, Munich, Germany, January 2000.

[HMO00c] J. Hagenauer, M. Moerz, and E. Offer. Recent progress in decoding with analog VLSI networks. In *Proc. 34th Conference on Information Sciences and Systems*, Princeton, NJ, USA, March 2000.

[HMS02] J. Hagenauer, M. Moerz, and A. Schaefer. Analog decoders and receivers for high speed applications. In *Proc. Int. Zurich Seminar on Broadband Communications*, pages 3.1–3.8, Zurich, Switzerland, February 2002.

[Hoc04] D. E. Hocevar. A reduced complexity decoder architecture via layered decoding of LDPC codes. In *Proc. IEEE Workshop on Signal Processing Systems*, pages 107–112, Austin, TX, USA, October 2004.

[HOMM99] J. Hagenauer, E. Offer, C. Méasson, and M. Moerz. Decoding and equalization with analog non-linear networks. *European Transactions on Telecommunications*, 10:659–680, Nov./Dec. 1999.

[HOP96] J. Hagenauer, E. Offer, and L. Papke. Iterative decoding of binary block and convolutional codes. *IEEE Transactions on Information Theory*, 42(2):429–445, March 1996.

[HW98] J. Hagenauer and M. Winklhofer. The analog decoder. In *Proc. IEEE Int. Symposium on Information Theory*, page 145, MIT, Cambridge, MA, USA, August 1998.

[IEE05] Low-density parity-check (LDPC) code proposal for IEEE 802.11n. Technical report, December 2005.

[JH99] G. Jeong and D. Hsia. Optimal quantization for soft-decision turbo decoder. In *Proc. IEEE Int. Conference on Vehicular Technology*, pages 1620–1624, Amsterdam, The Netherlands, September 1999.

[JKM00] H. Jin, A. Khandekar, and R. McEliece. Irregular repeat-accumulate codes. In *Proc. Int. Symposium on Turbo Codes and Related Topics*, pages 1–8, Brest, France, September 2000.

[JZ99] R. Johannesson and K. Sh. Zigangirov. *Fundamentals of Convolutional Coding*. IEEE Press, Picataway, NJ, USA, 1999.

[KDM+96] A. B. Kiely, S. J. Dolinar, R. J. McEliece, L. L. Ekroot, and W. Lin. Trellis decoding complexity of linear block codes. *IEEE Transactions on Information Theory*, 42(6):1687–1697, November 1996.

[KFL01] F. R. Kschischang, B. J. Frey, and H.-A. Loeliger. Factor graphs and the sum-product algorithm. *IEEE Transactions on Information Theory*, 47:498–519, February 2001.

[KLW06] F. Kienle, T. Lenigk-Emden, and N. Wehn. Fast convergence algorithm for LDPC codes. In *Proc. IEEE Int. Conference on Vehicular Technology*, Melbourne, Australia, May 2006.

[KS95] F. R. Kschischang and V. Sorokine. On the trellis structure of block codes. *IEEE Transactions on Information Theory*, 41(6):1924–1937, November 1995.

[LHC86] K. R. Lakshmikumar, R. A. Hadaway, and M. A. Copeland. Characterization and modeling of mismatch in MOS transistors for precision analog design. *IEEE Journal of Solid-State Circuits*, SC-21:1057–1066, 1986.

[LHL+99a] F. Lustenberger, M. Helfenstein, H.-A. Loeliger, F. Tarköy, and G. S. Moschytz. All-analog decoder for a binary (18,9,5) tail-biting trellis code. In *Proc. European Solid-State Circuits Conference*, pages 362–365, Duisburg, Germany, September 1999.

[LHL+99b] F. Lustenberger, M. Helfenstein, H.-A. Loeliger, F. Tarköy, and G. S. Moschytz. An analog VLSI decoding technique for digital codes. In *Proc. IEEE Int. Symposium on Circuits and Systems*, volume 2, pages 428–431, Orlando, FL, USA, May/June 1999.

[LL01] F. Lustenberger and H.-A. Loeliger. On mismatch errors in analog-VLSI error correcting decoders. In *Proc. IEEE Int. Symposium on Circuits and Systems*, volume IV, pages 198–201, Sydney, Australia, May/June 2001.

[LLHT98] H.-A. Loeliger, F. Lustenberger, M. Helfenstein, and F. Tarköy. Probability propagation and decoding in analog VLSI. In *Proc. IEEE Int. Symposium on Information Theory*, page 146, MIT, Cambridge, MA, USA, August 1998.

[Lus00] F. Lustenberger. *On the Design of Analog Iterative VLSI Decoders*. PhD thesis, ETH Zurich, Zurich, Switzerland, 2000.

[LWMM98] S. J. Lovett, M. Welten, A. Mathewson, and B. Mason. Optimizing MOS transistor mismatch. *IEEE Journal of Solid-State Circuits*, 33(1):147–150, January 1998.

[MA00] P. Moqvist and T. M. Aulin. Turbo-decoding as a numerical analysis problem. In *Proc. IEEE Int. Symposium on Information Theory*, page 485, Sorrento, Italy, June 2000.

[Mas74] J. L. Massey. Coding and modulation in digital communications. In *Proc. Int. Zurich Seminar on Digital Communications*, pages E2(1)–E2(4), Zurich, Switzerland, 1974.

[McE96] R. J. McEliece. On the BCJR trellis of linear block codes. *IEEE Transactions on Information Theory*, 42(4):1072–1092, July 1996.

[MGYH00] M. Moerz, T. Gabara, R. Yan, and J. Hagenauer. An analog 0.25$\mu$m BiCMOS tailbiting MAP decoder. In *Proc. IEEE Int. Solid-State Circuits Conference*, pages 356–357, San Francisco, CA, USA, February 2000.

[MHO00] M. Moerz, J. Hagenauer, and E. Offer. On the analog implementation of the APP (BCJR) algorithm. In *Proc. IEEE Int. Symposium on Information Theory*, page 425, Sorrento, Italy, June 2000.

[MN95] D. J. C. MacKay and R. M. Neal. Good codes based on very sparse matrices. In *Proc. 5th IMA Conf. Cryptography and Coding*, 1995.

[Moe99] M. Moerz. *Analog Decoders and their Implementation in VLSI*. Lehrstuhl für Nachrichtentechnik, Technische Universität München, Diplomarbeit, Munich, Germany, 1999.

[Moe01] M. Moerz. Analog decoding of high rate convolutional codes. In *Proc. 11th Joint Conference on Communication and Coding*, page 16, Bad Kleinkirchheim, Austria, March 2001. Lehrstuhl für Nachrichtentechnik, Technische Universität München.

[Moe02] M. Moerz. Analog sliding window decoding. In *Proc. Joint Workshop on Communications and Coding*, Barolo, Italy, November 2002. Lehrstuhl für Nachrichtentechnik, Technische Universität München.

[Moe03] M. Moerz. Decoding of convolutional codes using an analog ring decoder. In *Proc. 2nd Analog Decoding Workshop*, Zurich, Switzerland, September 2003.

[Moe04a] M. Moerz. Analog sliding window decoder core for mixed signal turbo decoder. In *Proc. Int. ITG Conference on Source and Channel Coding*, pages 63–70, Erlangen, Germany, January 2004.

[Moe04b] M. Moerz. Turbo decoding - analog or mixed signal? In *Proc. 3rd Analog Decoding Workshop*, Banff, Canada, June 2004.

[Moe05] M. Moerz. Quantization of soft-information in analog mixed-signal turbo decoding. In *Proc. 4th Analog Decoding Workshop*, Rennes, France, June 2005.

[Moe06] M. Moerz. Analog decoding method and decoder. US Patent 7071846, July 2006.

[MS93] T. W. Matthews and R. R. Spencer. An integrated analog CMOS Viterbi detector for digital magnetic recording. *IEEE Journal of Solid-State Circuits*, 28(12):1294–1302, December 1993.

[MS03] M. M. Mansour and N. R. Shanbhag. High-throughput LDPC decoders. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 11(6):976–996, December 2003.

[MSO01] M. Moerz, A. Schaefer, and E. Offer. Analog decoding of high rate tailbiting codes using the dual trellis. In *Proc. IEEE Int. Symposium on Information Theory*, page 331, Washington D.C., USA, June 2001.

[MSOH01] M. Moerz, A. Schaefer, E. Offer, and J. Hagenauer. Analog decoders for high rate convolutional codes. In *Proc. IEEE Information Theory Workshop*, pages 128–130, Cairns, Australia, September 2001.

[MV91] K. Meyberg and P. Vachenauer. *Höhere Mathematik 2*. Springer-Verlag, Berlin Heidelberg, Germany, 1991.

[MW86] H. H. Ma and J. K. Wolf. On tail biting convolutional codes. *IEEE Transactions on Communications*, 34(2):104–111, February 1986.

[MW01] H. Michel and N. Wehn. Turbo-decoder quantization for UMTS. *IEEE Communications Letters*, 5(2):55–57, February 2001.

[MWW00] H. Michel, A. Worm, and N. Wehn. Influence of quantization on the bit-error performance of turbo-decoders. In *Proc. IEEE Int. Conference on Vehicular Technology*, pages 581–585, Tokyo, Japan, May 2000.

[NWGS04] N. Nguyen, C. Winstead, V. C. Gaudet, and C. Schlegel. A 0.8V CMOS analog decoder for an (8,4,4) extended Hamming code. In *Proc. IEEE Int. Symposium on Circuits and Systems*, pages 1116–1119, Vancouver, Canada, May 2004.

[Pro95] J. G. Proakis. *Digital Communications*. McGraw-Hill, New York, USA, 3rd edition, 1995.

[Rob94] P. Robertson. Illuminating the structure of code and decoder of parallel concatenated recursive systematic (turbo) codes. In *Proc. IEEE Global Communications Conference*, volume 3, pages 1298–1303, San Francisco, CA, USA, Nov./Dec. 1994.

[RU01] T. J. Richardson and R. Urbanke. The capacity of low-density parity-check codes under message passing decoding. *IEEE Transactions on Information Theory*, 47:599–618, February 2001.

[RVH95] P. Robertson, E. Villebrun, and P. Hoeher. A comparison of optimal and sub-optimal MAP decoding algorithms operating in the log domain. In *Proc. IEEE Int. Conference on Communications*, volume 2, pages 1009–1013, 1995.

[Sch05] A. Schaefer. *Insights and Analysis of Analog and Iterative Decoding*. PhD thesis, Technische Universität München, Munich, Germany, 2005.

[SGPMM06] J. Soler-Garrido, R. J. Piechocki, K. Maharatna, and D. McNamara. MIMO detection in analog VLSI. In *Proc. IEEE Int. Symposium on Circuits and Systems*, Island of Kos, Greece, May 2006.

[Sha48] C. E. Shannon. A mathematical theory of communication. *Bell Syst. Tech. J.*, 27:379–423 (Part I) and 623–656 (Part II), July and October 1948.

[SJM94] M. S. Shakiba, D. A. Johns, and K. W. Martin. General approach to implementing analogue Viterbi decoders. *Electronic Letters*, 30(22):1823–1824, October 1994.

[SJM98] M. S. Shakiba, D. A. Johns, and K. W. Martin. BiCMOS circuits for analog Viterbi decoders. *IEEE Transactions on Circuits and Systems—Part II: Analog and Digital Signal Processing*, 45(12):1527–1537, December 1998.

[SS96] M. Sipser and D. A. Spielman. Expander codes. *IEEE Transactions on Information Theory*, 42(6):1710–1722, November 1996.

[SSM+03a] A. Schaefer, A. Sridharan, M. Moerz, J. Hagenauer, and D. J. Costello, Jr. Analog rotating ring decoder for an LDPC convolutional code. In *Proc. IEEE Information Theory Workshop*, pages 226–229, Paris, France, April 2003.

[SSM+03b] A. Schaefer, A. Sridharan, M. Moerz, J. Hagenauer, and D. J. Costello, Jr. Analog rotating ring decoder for an LDPC convolutional code. In *Proc. 2nd Analog Decoding Workshop*, Zurich, Switzerland, September 2003.

[SvT79] G. Solomon and H. C. A. van Tilborg. A connection between block and convolutional codes. *SIAM Journal on Applied Mathematics*, 37(2):358–369, October 1979.

[Tan81] R. M. Tanner. A recursive approach to low complexity codes. *IEEE Transactions on Information Theory*, IT-27:533–547, September 1981.

[tB99] S. ten Brink. Convergence of iterative decoding. *Electronic Letters*, 35(10):806–808, May 1999.

[tB00] S. ten Brink. Design of serially concatenated codes based on iterative decoding convergence. In *Proc. Int. Symposium on Turbo Codes and Related Topics*, pages 319–322, Brest, France, September 2000.

[tB01] S. ten Brink. Convergence behavior of iteratively decoded parallel concatenated codes. *IEEE Transactions on Communications*, 49(10):1727–1737, October 2001.

[TH02] M. Tüchler and J. Hagenauer. EXIT charts of irregular codes. In *Proc. Conf. Information Sciences and Systems*, Princeton, NJ, USA, March 2002.

[Tsi99] Y. Tsividis. *Operation and Modelling of The MOS Transistor*. McGraw-Hill, Boston, MA, USA, 2nd edition, 1999.

[Vei02] A. Veinblat. Analog convolutional decoder. Master's thesis, Technion, Israel Institute of Technology, May 2002.

[VGN+05] D. Vogrig, A. Gerosa, A. Neviani, A. Graell i Amat, G. Montorsi, and S. Benedetto. A 0.35$\mu$m CMOS analog turbo decoder for the 40-bit, rate 1/3, UMTS channel code. *IEEE Journal of Solid-State Circuits*, 40(3):753–762, March 2005.

[Vit67] A. J. Viterbi. Error bounds for convolutional codes and an asymptotic optimum decoding algorithm. *IEEE Transactions on Information Theory*, 13(4):260–269, April 1967.

[Vit98] A. J. Viterbi. An intuitive justification and a simplified version of the MAP decoder for covolutional codes. *IEEE Journal on Selected Areas in Communications*, 16(2):260–264, February 1998.

[WBRC98] C. Weiß, C. Bettstetter, S. Riedel, and D. C. Costello, Jr. Turbo decoding with tailbiting trellises. In *Proc. URSI Int. Symposium on Signals, Systems and Electronics*, pages 343–348, Pisa, Italy, September/October 1998.

[WCC+05] I. C. Wey, L. H. Chang, Y. G. Chen, S. H. Chang, and A. Y. Wu. A 2Gb/s high-speed scalable shift-register based on-chip serial communication design for SoC applications. In *Proc. IEEE Int. Symposium on Circuits and Systems*, pages 1074–1077, Kobe, Japan, May 2005.

[WDK+01] C. Winstead, J. Dai, W. J. Kim, S. Little, Y.-B. Kim, C. Myers, and C. Schlegel. Analog MAP decoder for (8,4) Hamming code in subthreshold CMOS. In *Proc. Advanced Research in VLSI*, pages 132–147, Salt Lake City, UT, USA, March 2001.

[WDL+01] C. Winstead, J. Dai, S. Little, C. Myers, C. Schlegel, Y.-B. Kim, and W. J. Kim. Analog MAP decoder for (8,4) Hamming code in subthreshold CMOS. In *Proc. IEEE Int. Symposium on Information Theory*, page 330, Washington D.C., USA, June 2001.

[WDY+04] C. Winstead, J. Dai, S. Yu, C. Myers, R.R. Harrison, and C. Schlegel. CMOS analog MAP decoder for (8,4) Hamming code. *IEEE Journal of Solid-State Circuits*, 39(1):122–131, January 2004.

[Wei02] C. Weiß. *Error Correction with Tail-Biting Convolutional Codes*. PhD thesis, Technische Universität München, Munich, Germany, 2002.

[Wib96] N. Wiberg. *Codes and Decoding on General Graphs*. PhD thesis, Univ. Linköping, Linköping, Sweden, 1996.

[Win98] M. Winklhofer. *Analoge Decodierung von Block- und Faltungscodes*. Lehrstuhl für Nachrichtentechnik, Technische Universität München, Diplomarbeit, Munich, Germany, 1998.

[Win04] C. Winstead. *Analog Iterative Error Control Decoders*. PhD thesis, Univ. of Alberta, Edmonton (Alberta), Canada, 2004.

[WLK95] N. Wiberg, H.-A. Loeliger, and R. Kötter. Codes and iterative decoding on general graphs. *European Transactions on Telecommunications*, 6:513–525, Sept./Oct. 1995.

[WNGS06] C. Winstead, N. Nguyen, V. C. Gaudet, and C. Schlegel. Low-voltage CMOS circuits for analog iterative decoders. *IEEE Transactions on Circuits and Systems—Part I: Fundamental Theory and Applications*, 53(4):829–841, April 2006.

[WW99] Y. Wu and B. D. Woerner. The influence of quantization and fixed point arithmetic upon the BER performance of turbo codes. In *Proc. IEEE Int. Conference on Vehicular Technology*, pages 1683–1687, Houston, TX, USA, May 1999.

[XVG+02] A. Xotta, D. Vogrig, A. Gerosa, A. Neviani, A. Graell i Amat, G. Montorsi, M. Bruccoleri, and G. Betti. An all-analog CMOS implementation of a turbo decoder for hard-disk drive read channels. In *Proc. IEEE Int. Symposium on Circuits and Systems*, pages V–69–V–72, Scottsdale, AZ, USA, May 2002.