Steffen Kunze

On the Design and Implementation of Multi-Mode Channel Decoders

Beiträge aus der Informationstechnik

Mobile Nachrichtenübertragung
Nr. 62

Steffen Kunze

On the Design and Implementation of Multi-Mode Channel Decoders

VOGT

Dresden 2013

Bibliografische Information der Deutschen Bibliothek
Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.ddb.de abrufbar.

Bibliographic Information published by the Deutsche Bibliothek
The Deutsche Bibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibiograpic data is available in the internet at http://dnb.ddb.de.

Zugl.: Dresden, Techn. Univ., Diss., 2013

Die vorliegende Arbeit stimmt mit dem Original der Dissertation „On the Design and Implementation of Multi-Mode Channel Decoders“ von Steffen Kunze überein.

Gesetzt vom Autor

ISBN 978-3-938860-66-3

Jörg Vogt Verlag
Niederwaldstr. 36
01277 Dresden
Germany

Phone: +49-(0)351-31403921
Telefax: +49-(0)351-31403918
e-mail: info@vogtverlag.de
Internet : www.vogtverlag.de

TECHNISCHE UNIVERSITÄT DRESDEN

ON THE DESIGN AND IMPLEMENTATION OF MULTI-MODE CHANNEL DECODERS

Steffen Kunze

der Fakultät Elekrotechnik und Informationstechnik
der Technischen Universität Dresden

zur Erlangung des akademischen Grades

Doktoringenieur
(Dr.-Ing.)

genehmigte Dissertation

Vorsitzender:	Prof. Dr.-Ing. Eduard Jorswieck, TU Dresden
Gutachter:	Prof. Dr.-Ing. Dr. h.c. Gerhard Fettweis, TU Dresden
	Prof. Luca Fanucci, Uni Pisa
Tag der Einreichung:	06.05.2013
Tag der Verteidigung:	20.08.2013

Acknowledgements

I would like to express my sincerest gratitude to Prof. Fettweis for supervising this thesis and to my colleagues at the chair, especially the current and former members of Dr. Matus' group, for providing a brilliant working environment.

Abstract

In modern mobile communications systems the designer has the task of balancing user requirements of high data throughputs with the cost concerns of low power consumption and implementation costs. One important aspect of such systems is the forward error correction (FEC) coding employed to ensure reliable communication. In the receiver side of a mobile communications system-on-a-chip (SoC) the FEC decoding contributes a significant part of design complexity. Unfortunately, there exist a multitude of FEC schemes which are deployed in today's communication standards to deal with different application scenarios and these schemes often are implemented in separate IP blocks in a SoC. In accordance with the above mentioned goals of low power and slim implementation it is attractive to try and combine the decoding functionality for several FEC schemes into one IP block.

This work presents a framework of methods and rules that can be applied to execute such a merging of several decoding schemes. One part of this is analyzing target algorithms in their two main aspects communication and processing and the other applying findings from this analysis on different hierarchical levels to determine the applicable methods of decoder merging.

Furthermore, the proposed concepts are put into practice through the design of a multi-mode FEC decoder capable of decoding convolutional, turbo and LDPC codes to demonstrate application of the merging methods in different design steps to reach the final decoder realization. At a post-synthesis area of 0.77 mm^2 the decoder reaches throughputs of 23.1 to 86.4 Mbps at a 200 MHz clock frequency, showing that it is on par with published works and therefore proving that the presented method can be used to produce state-of-the-art results. The decoder has also been implemented in silicon using a 65nm TSMC process, providing useful measurements to verify simulation results.

Another main topic of this work is the assessment of merging efficiency, a point that has been neglected so far in literature. To this end, single-mode decoders specialized on decoding just one type of codes - either Viterbi, Turbo or LDPC - are derived from the multi-mode architecture. Comparisons between multi-mode and single-mode decoders indicate that the former has

a smaller overall area but suffers from an increase in dissipated power.

Contents

List of Figures

List of Tables

Nomenclature

ACS	*Add-compare-select*
AGU	*Adress generation unit*
ALU	*Arithmetic logic unit*
ASIC	*Application-specific integrated circuit*
ASIP	*Application-specific instruction set processor*
BCJR	*Bahl-Cocke-Jelinek-Raviv*
BER	*Bit error rate*
BM	*Branch metric*
CC	*Convolutional code*
CN	*Check node*
CRC	*Cyclic redundancy check*
CTC	*Convolutional turbo code*
DSP	*Digital signal processor*
DVB-S	*Digital video broadcast-satellite*
FEC	*Forward error correction*
FIFO	*First in first out*
FPGA	*Field programmable gate array*
FU	*Functional unit*
GPP	*General-purpose processor*
LDPC	*Low-density parity-check code*
LLR	*Log likelihood ratio*
LUT	*Look-Up table*
MAP	*Maximum a-posteriori propability*
MIMO	*Multiple Input Multiple Output*
MPSoC	*Multi-processor system-on-a-chip*
NoC	*Network-on-chip*

PE	*Processing element*
PLL	*Phase-locked loop*
QC	*Quasi-cyclic*
RISC	*Reduced instruction set computing*
SIMD	*Single instruction multiple data*
SISO	*Soft input soft output*
SM	*State metric*
STA	*Synchronous transfer architecture*
TPMP	*Two-phase message passing*
TSMC	*Taiwan Semiconductor Manufacturing Company*
VA	*Viterbi algorithm*
VLIW	*Very long instruction word*
VLSI	*Very large scale integration*
VN	*Variable node*
$\alpha_i(S_k)$	Forward recursion state metric for trellis state S_k in trellis step i
$\beta_i(S_k)$	Backward recursion state metric for trellis state S_k in trellis step i
β_O	Min-sum normalization offset
$\gamma_i(S_{k,i}, S_{j,i-1})$	Branch metric for transition from trellis state S_k in trellis step i to state S_j in step j
$\Lambda(u_i)$	LLR corresponding to source bit i
$\lambda_{jk}(i)$	Branch metric for transition from trellis state S_j to state S_k in trellis step i
$\Lambda_k(i)$	Path metric of trellis state S_k in trellis step i
$\mathbf{C}$	Check nodes in a tanner graph
$\mathbf{u}$	Source symbols
$\mathbf{V}$	Variable nodes in a tanner graph
$\mathbf{v}$	Coded symbols
$\mathbf{y}$	Received symbols
$\mathbf{z}$	Submatrix size of a QC-LDPC code
ν	Area efficiency ratio
B	Length of quantized receive symbol
$d_k(i)$	Decision bit generated at trellis state S_k in trellis step i
E_r	Energy efficiency measure in nJ/bit
f_{clock}	Clock frequency
G	Generator matrix

H	Parity-check matrix
K	Length of source symbol vector
k	Constraint length of a conv. code
m_s	Memory depth of a conv. encoder
N	Length of code symbol vector
n_b	Number of submatrices in a row of H
n_m	Number of submatrices in a column of H
N_q	Number of quantization bits
N_{iter}	Number of decoding iterations
N_{sub}	Number of submatrices in H
O	Area overhead ratio
P_N	Number of processing nodes in a core
P_P	Internal parallelism of a processing node
R	Code rate
R_s	Area sharing ratio
S	Area savings ratio
S_k	Trellis state representing encoder state k
T_{LDPC}	Throughput in LDPC decoding mode
T_{Turbo}	Throughput in turbo decoding mode
$T_{Viterbi}$	Throughput in Viterbi decoding mode
z_{jk}	Expected encoder output for transition from trellis state S_j to state S_k

1 Introduction

1.1 Motivation

Modern communication systems are characterized by the need to transmit large amounts of data at high data rates. To ensure reliable communication, in most cases it is beneficial to employ forward error correction (FEC) techniques to avoid costly retransmission and optimize the efficiency of radio resource usage. Throughout the history of information theory and communications engineering, a large number of different FEC schemes have been developed. Well known examples are BCH codes, Reed-Solomon codes, convolutional codes (CCs), turbo codes or the recently more popular low-density parity-check (LDPC) codes.
Unfortunately, none of these coding schemes has been shown to be a catch-all solution to cover the wide range of communication scenarios that exist. This leaves us in a situation where modern communication standards use different coding schemes and code parameters from one standard to another because while one coding scheme might excel in one application case, it may be inefficient in another. It is even quite common for one communication standard to employ more than one type of channel coding algorithm in its various specified scenarios, e.g. one for a control channel and another for a data channel. The IEEE 802.15 and LTE standards for example both use convolutional turbo codes (CTC) as well as CCs. The IEEE 802.15e 'Wimax mobile' standard even has an additional option for LDPC codes.
The system designer who wants to implement the communications system in hardware now has to accommodate for all the required coding types and parameter sets. The main concern here is the decoding part of the signal processing chain, since it is far more computationally complex than the encoding. So complex even, that it is often infeasible to implement the decoding algorithms on standard general-purpose processors or DSPs. Instead, FEC decoders are generally implemented in hardware as specialised IP blocks. Traditionally, this also meant having a dedicated IP core for every coding scheme, which of course uses valuable chip resources and increases system complexity. Recently however, there has been some effort to merge decoder

architectures for different algorithms. This type of approach gives the system designer the option to reduce the number of IP blocks in the System-on-Chip (SoC) and avoid having to spend chip area for specialised IP that might only be used a small part of the time. Some notable results have been reported for combining LDPC and CTC decoders in [Nae08], [Gen10] and [Sun08]. In [All08] LDPC, CTC and CC decoding is combined.

However, all publications on this topic so far are only focused on particular implementations of the decoder merging problem. As of yet there has been no attempt to analyze it in a more general way and the published work on the methodology of combining decoding algorithms and evaluating the results remains sparse. A reason for this is that in general, it is difficult and not very meaningful to compare architectures that are often implemented on different technologies with different design goals in mind, making it hard to accurately assess them. In this work we want to take a more methodological approach to this problem. Merging the hardware implementations of channel decoders shall be analyzed from a more theoretical point of view as well as through actual implementation results. The latter shall be achieved by following an approach that not only gives us a multi-mode FEC core but also some benchmark implementations to allow a transparent and fair comparison to answer the question whether it is actually efficient to merge decoders or if it would be advantageous to keep dedicated decoders over a multi-mode solution.

The basic idea here is to take a fixed set of technology parameters and take a common design framework to design and implement a decoder capable of processing several different decoding algorithms with state-of-the-art performance. Using this main decoder as reference point, it is then possible to design and implement several separate 'benchmark' decoders that are dedicated to one of these algorithms each. With this we can collect the data that should prove as a reliable source for drawing conclusions on the effects of merging decoders and answering the open questions we mentioned thus far.

1.2 Organization of the thesis

In **chapter 2** an introduction to some basics of forward error correction is given and the decoding algorithms which will later become the focus of the hardware implementation are introduced.

Chapter 3 categorizes methods for the analysis of algorithms that should be merged in hardware and details design methods that can be used to execute the merging of architectures. This is accompanied by an overview of the state-of-the-art in this area.

Chapter 4 shows the application of the methods defined in chapter 3 through a realization of a multi-mode channel decoder. The decoder architecture will be presented together with some results of the implementation.
Chapter 5 is dedicated to the description of the ’benchmark’ single-mode decoders based on the architecture from chapter 4 and their implementations.
Chapter 6 discusses the results gained by merging the decoders and gives an evaluation of the implementation results.
Chapter 7 contains concluding remarks and an outlook into possible future work.

2 Channel decoding and it's implementation aspects

This chapter gives a brief introduction to the topic of forward error correction, or channel coding, as it is also commonly called. Three common channel coding schemes used in modern communication systems - convolutional codes, turbo codes and low-density parity-check codes - are explained briefly with an overview of some decoding algorithms as well as pointing out some challenges that accompany a hardware implementation of these algorithms.

2.1 Channel coding basics

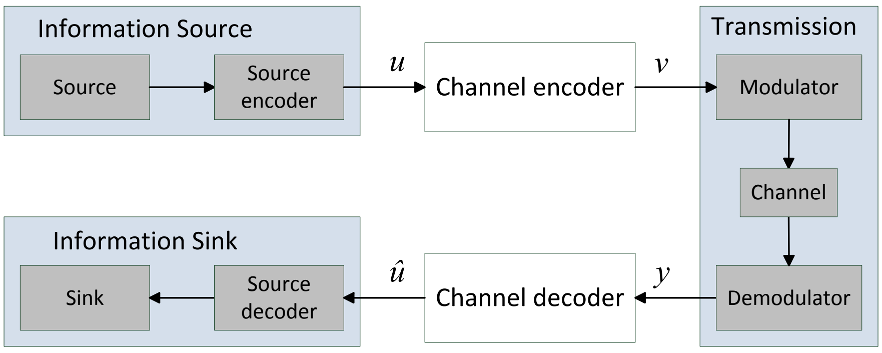

Figure 2.1: System model

In figure 2.1 you can see the general model of the kind of communication system we assume in our work. Binary source data $\mathbf{u}$ is encoded with a channel code of code rate R to improve the system performance under faulty transmission conditions through the exploitation of added redundancy. The encoded symbols $\mathbf{v}$ are then modulated and transmitted over a noisy channel. On the receiver side, these steps are reversed. The demodulator transforms the transmitted signal back into a discrete sequence of received symbols $\mathbf{y}$. Based on these symbols, the channel decoder generates an estimate $\hat{\mathbf{u}}$ of the source data and passes it to the information sink.

2.1.1 Block codes and convolutional codes

Channel codes are separated into two categories based on the way the code sequence is generated. A code is called a **block code**, when an information sequence $\mathbf{u} = (u_0, u_1, ..., u_K)$ of length K is transformed into a code sequence $\mathbf{v} = (v_0, v_1, ..., v_N)$ of length N by multiplying it with a matrix $\mathbf{G}$, the so-called generator matrix.

$$G \cdot u = v \tag{2.1}$$

Thus, redundancy is added to the original information that makes it possible to detect and/or correct errors that occurred during transmission. The resulting code rate is given by $R = N/K$. Additionally, there exists the parity-check matrix $\mathbf{H}$ that can be used to determine, whether a received sequence is a valid codeword. For this to be true it has to fulfill the equation

$$H \cdot v^T = 0. \tag{2.2}$$

In the case of the block codes, the whole code sequence can be generated in one step by carrying out the matrix multiplication.

Convolutional codes (CCs) take a different approach. The input information sequence is processed in a serial fashion, symbol by symbol. Figure 2.2 pictures the encoding process.

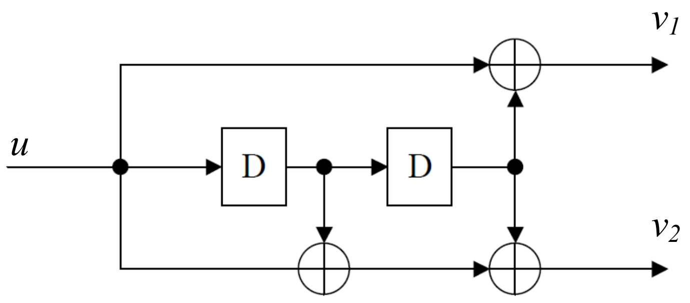

Figure 2.2: Convolutional encoder

The information sequence $\mathbf{u}$ is shifted into a set of $\mathbf{m_s}$ memory elements. The feedforward connections to the outputs or feedback connections to the memory elements can be represented

by a generator polynomial $\mathbf{G}$. In our case, the polynomial corresponding to output $\mathbf{v_1}$ would be $G_1 = 1 \cdot x^{-2} + 0 \cdot x^{-1} + 1 \cdot x$, which often gets shortened to $(1, 0, 1)$ respectively the octal equivalent of just (5). The whole encoder is then described by $\mathbf{G} = (G_1, G_2) = (5, 7)$.
The depth of the encoder memory of a CC is also referred to as constraint length k, which is defined as $k = m_s + 1$.
The two basic types of channel codes mentioned above are the basis for most coding schemes that have been developed so far. To increase the code performance, channel codes are often concatenated in either a serial or parallel fashion.

2.1.2 Challenges in channel decoding

The main challenge of implementing channel decoding algorithms in hardware is to provide good error correction performance and high data throughput while keeping chip area and power consumption low. Unfortunately, these goals contradict each other, as high throughput and good error correction performance are correlated with rising hardware complexity. The system designer now has the challenging task to find a trade-off between acceptable performance loss and hardware cost to implement the decoding. Three important trade-offs are listed in the following:
Accuracy-complexity-tradeoff: The goal of the decoding step is to find the codeword $\mathbf{v}$ that has the highest probability of being transmitted based on the received vector $\mathbf{r}$. The optimal result is achieved by performing a maximum-likelihood search over the whole codeword space, but the high computational complexity required makes this approach impractical for implementation in hardware. This led to the development of sub-optimal decoding algorithms, that offer greatly reduced computational complexity at the price of slightly lower communications performance. Still, channel decoding remains as one of the most computationally complex parts of the receiver signal processing chain in high-throughput communication systems. Some choice sub-optimal algorithms are going to be discussed in the next sections.
Another aspect of decoding where this performance-complexity trade-off comes into play is the quantization of the received symbols. Instead of performing a "hard" decision between the binary values 1 and 0, the receiver can also provide them in a higher-valued representation of N_q bits. This gives us an additional measure of the reliability of the estimation and can be exploited to get a better decoding result. Again, this comes with the price of a higher implementation cost, since now all operations have to be performed on values of N_q bits. Every decoding algorithm has a point at which the performance gains from increasing N_q become negligible compared to the additional complexity, which should be taken into account as a constraint for the system design.

Throughput-complexity-tradeoff: To speed up decoding to satisfy the requirements of modern communications systems, there are two methods we can employ: increasing clock frequency of the decoder and parallelizing the decoding. The former increases hardware complexity a bit by added pipeline stages but after a certain point a rise in clock frequency results in a disproportionate rise in power consumption. Parallelization obviously increases the chip area, but has the potential to be more power-efficient. A challenge is finding algorithms that are fit for high degrees of parallelization to achieve very high throughput.
Flexibility-complexity-tradeoff: When implementing a channel decoder, the design is usually limited to a certain set of code parameters and the complexity of the design is directly related to the size of this parameter set. Supporting only a very small set of parameters allows for a heavily specialized and very efficient implementation, but adding support for more and more parameters increases complexity. The case analyzed in this thesis can be even more complex, since not only different code parameters but also different code types are taken into consideration.

These criteria are the major deciding factor when selecting decoding algorithms for hardware implementation. The following sections briefly explain suboptimal decoding algorithms for three channel coding types that are widely used in the area of mobile communications and take a look at some specific implementation aspects for each of them.

2.2 Convolutional Decoding - the Viterbi algorithm

The Viterbi algorithm (VA) [Vit67] is a well-known algorithm to non-iteratively decode CCs. It accumulates metrics along paths in the trellis diagram based on the received symbols and selects the path with the best metric, representing the most likely code word.

The VA is traditionally divided into three parts: branch metric computation (BMC), state metric computation (SMC), and traceback (TB). In the BMC, for every trellis step i the distance $\lambda_{jk}(i) = |y(i) - z_{jk}|^2$ between received input symbols $\mathbf{y}(i) = (y_0, y_1, ..., y_b)$ and expected values $\mathbf{z_{jk}}$ is calculated, where $\mathbf{z_{jk}}$ is the expected encoder output for a transition from trellis states S_j to S_k. This distance can be calculated as sum of the distances $\lambda_{jk,b}$ for each of the individual symbol elements y_b. These are quantized as unsigned integer soft values ranging from zero to a maximum integer value y_{max}, so $\lambda_{jk,b}$ is essentially the distance of y_b to either zero or y_{max}, which is equivalent to either the input value itself or its bitwise negations, depending on z_{jk}.

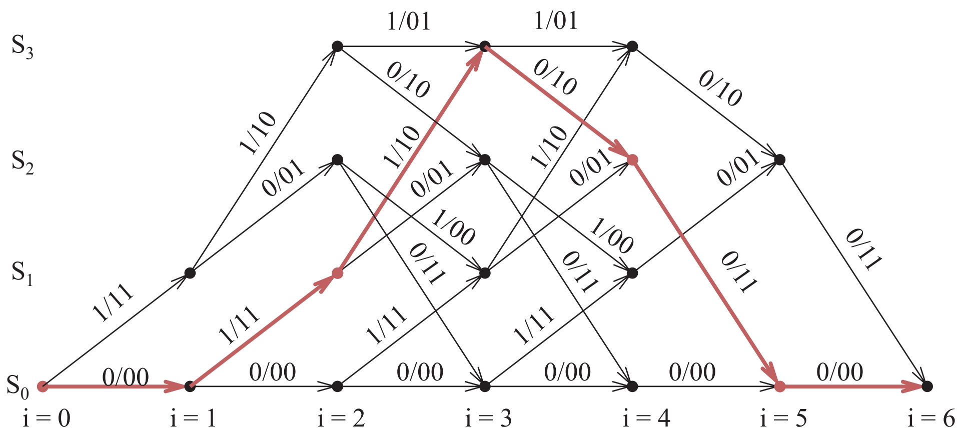

Figure 2.3: Trellis graph corresponding to the state transitions of the encoder from 2.2 with the highlighted path representing the input sequence $\{0, 1, 1, 0, 0, 0\}$.

The accumulated sum for one input symbol becomes:

$$\lambda_{jk}(i) = \sum_{b=1}^{B} \lambda_b \quad \text{with} \quad \lambda_b = \begin{cases} y_b(i) & \text{if} \quad z_{jk,b} = 0 \\ \bar{y_b}(i) & \text{if} \quad z_{jk,b} = 1 \end{cases} \tag{2.3}$$

where B is the number of elements in an input vector (i.e. $B = 2$ for a rate-1/2 code) and $\bar{y_b}$ is the bitwise negation of y_b.

The resulting metrics $\lambda_{jk}(i)$ are used in the SMC to update and select the new path metrics in a recursive Add-Compare-Select (ACS) operation:

$$\Lambda_k(i) = \min_j(\Lambda_j(i-1) + \lambda_{jk}(i)). \tag{2.4}$$

This selection equates to taking a decision on whether a 0 or 1 was likely encoded in step i. These 'decision bits' $d_k(i)$ from the SMC are later used to trace back the maximum-likelihood path and form the decoded codeword after all state transitions along the trellis have been calculated.

Implementation concerns

To implement a given algorithm in hardware always means looking for solutions to the already mentioned tradeoffs between performance and costs. Let us look at some interesting aspects of the VA in regard to these constraints. To speed up the decoding process, the most important tool is algorithm parallelization, which can happen on various hierarchical levels. Bit-level optimizations to VA implementations were presented in [Fet91]. Another low-level parallelization option for trellis-based codes in general is the folding of several trellis steps into one [Fet89]. To combine two steps, the ACS has to be performed over 4 path metrics in one clock cycle, which is also called a radix-4 ACS in [Bla92]. On the algorithmic level, a very straightforward method is to process state metric computations for several trellis states in parallel. Because often codes with a high number of trellis states (e.g. 64) are used, this is already a powerful tool and makes it fairly easy to reach high throughputs of several hundreds of Megabits per second (Mbps). To achieve even higher performance, the so-called windowing technique is used [Bla96]. The codeword is split into several windows, which are again processed in parallel. Since windows other than the first start the processing in an unknown state, they have to be initialised first, which introduces additional computations. Due to its non-iterative nature, the VA is the least computationally complex and data-transfer intensive algorithm out of the ones shown here and since it has been used for about 50 years now its implementation has been studied very extensively.

2.3 Turbo decoding - the BCJR algorithm

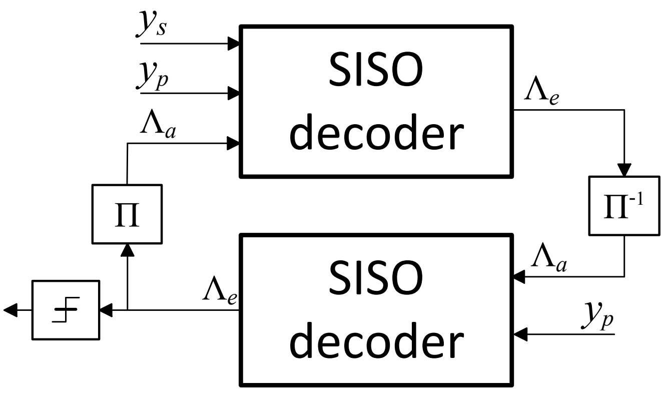

Figure 2.4: Two SISO decoders coupled by (de)interleavers make up a turbo decoder

Convolutional turbo codes (CTCs) are a class of parallely concatenated CCs. The most common type, which we will be discussing, consists of two systematic rate-$\frac{1}{2}$ CCs with an interleaver inbetween the encoders and one of the systematic encoder outputs punctured [Ber93]. Turbo decoding is performed iteratively in a loop between two component decoders which use the BCJR algorithm [BCJR74] to decode the individual CCs and exchange the generated reliability info (see fig. 2.4). In contrast to the VA, the BCJR algorithm decodes CCs with the goal of creating soft outputs that include information about bit reliability. Therefore, state metrics are calculated not only in a forward recursion similar to the VA but also in a backwards recursion. The result of the two is combined into the a posteriori probability $\Lambda(u_i)$ for an information bit u_i. Forward and backward recursions are defined as follows:

$$\alpha_i(S_k) = \overset{*}{\max_j}(\alpha_{i-1}(S_k) + \gamma_i(S_{k,i}, S_{j,i-1})) \tag{2.5}$$

$$\beta_i(S_k) = \overset{*}{\max_j}(\alpha_{i+1}(S_k) + \gamma_i(S_{k,i}, S_{j,i+1})) \tag{2.6}$$

where γ are the branch metrics and

$$\overset{*}{\max}(a,b) = max(a,b) - \ln(1 + e^{-|a-b|}). \tag{2.7}$$

The max-star operator can be approximated by $max()$ with a small loss of precision. This is then known as the max-log-MAP algorithm [Rob97].
After calculating the state metrics, forward and backward metrics are then combined to generate the LLR for this bit:

$$\Lambda(u_i) = \overset{*}{\max_{u_i=1}}(\alpha_{i-1} + \gamma_i + \beta_i) - \overset{*}{\max_{u_i=0}}(\alpha_{i-1} + \gamma_i + \beta_i) \tag{2.8}$$

With these LLRs the extrinsic information to be exchanged between the component decoders is extracted:

$$\Lambda_e(u) = \Lambda(u) - (\Lambda_a(u) + \Lambda(r_s) + \Lambda(r_p)) \tag{2.9}$$

One turbo decoding iteration is made up of two such symbol-by-symbol maximum-a-posteriori (MAP) decoding processes, one for each component CC. They are also called sub-iterations.

Implementation concerns

The fact that we have to process a forward and a backward recursion and afterwards perform a LLR combination step in each of two sub-iterations per turbo iteration makes turbo decoding a very computationally complex task. Generally we can apply the same parallelization techniques as with the VA, but parallelizing BCJR takes more effort, since the component codes usually have only 8 or 16 trellis states. Besides parallel state metric computation, forward and backward recursion can be executed in parallel to speed up decoding. This basic scheme is referred

to as 'sliding window' decoding [Ben96]. Due to the iterative nature of this algorithm this is still not enough to reach high throughputs, so extensive windowing is required. However, this can conflict with the interleaving that has to be performed inbetween subiterations. Windowing in this case means that a codeword is divided into P blocks that are decoded on P MAP decoders in parallel. As mentioned earlier, some initialization is needed for these blocks and there are different approaches to this problem such as next iteration initialization or the XMAP [Daw93][Wor00].
Another potentially complex part of a turbo decoder is the interleaver between the two subiterations that is used to reduce dependencies between the encoded data streams. Parallel memory accesses mapped to the same memory bank by the interleaver can cause conflicts and make it necessary to use algorithms that are contention-free up to a certain level of parallelism. If the necessary addresses cannot be generated on-the-fly at runtime, they need to be precomputed and stored in memories, which can become quite large for long codewords. This means it is desirable to use algorithms that are both contention-free and easy to process. Polynomial-based algorithms with such properties have been proposed in [Tak06] and [Ber04].

An important method to avoid redundant computation is to exactly control the number of executed turbo iterations. The number of iterations necessary before the decoded codeword converges depends on channel conditions and also differs from codeword to codeword, so performing a cyclic redundancy check (CRC) to test for successful decoding after each (sub-)iteration can prevent execution of redundant iteration cycles where the result is not improved any more [Shi99].

2.4 LDPC decoding - the Min-Sum algorithm

LDPC codes are block codes with typically very sparse parity-check matrices that offer similar error correction capabilities as turbo codes. A common practical suboptimal approach to decode LDPC codes is the offset min-sum approximation [Che05] of Gallager's optimal Belief Propagation algorithm [Gal63] which operates on a graphical representation of LDPC codes, the Tanner graph [Tan81]. The graph is an elegant alternate way of visualizing the parity-check matrix: M Check nodes (CNs) represent the rows (or parity-check equations), N variable nodes (VNs) represent the columns (or code bits) and edges between nodes represent the 1-entries of the matrix.

For Min-Sum decoding, the reliability information from the recieved channel values (repre-

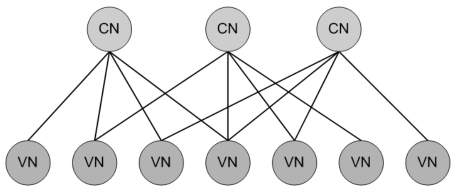

Figure 2.5: Tanner graph of an irregular LDPC Code

sented as log-likelihood ratio (LLR)) is propagated iteratively between the nodes and checked against the parity equations. This works in four basic steps:
1) **Initialization**: Graph edges are initialized to the received channel values y.
2) **CN update**: The CNs perform a parity check over the connected VNs $\mathbf{V}$ and propagate the result together with the corresponding LLRs back to the VNs (excluding intrinsic information from the respective nodes). The update from a CN C_m to a VN V_n can be expressed as:

$$L_{mn} = \left(\prod_{n' \in V_{m \setminus n}} \text{sign}(L_{n'm}) \right) \cdot \max(\min_{n' \in V_{m \setminus n}} |L_{n'm}| - \beta_O, 0). \tag{2.10}$$

β_O serves as normalization offset to prevent performance degradation, the max function around the min-term and 0 simply indicates β_O shall not be used on values smaller than itself.
3) **VN update**: The VNs collect the LLRs from the connected CNs C and updates them with the new values (again, minus intrinsic information):

$$L(_{nm}) = y_n + \sum_{m' \in C_{n \setminus m}} L(r_{m'n}) \tag{2.11}$$

4) **Decision**: Steps 2 and 3 are repeated iteratively until a stopping criterion is reached. Now the soft outputs of step 3 (this time including intrinsic information) are subjected to hard decision.

In the original two-phase message passing (TPMP) schedule first all check node updates are calculated and afterwards all variable node updates. It has been shown that convergence can be sped up by instead updating all variable nodes connected to a certain check node directly after that node's update [Man03]. This technique is usually called Layered Decoding.
In general it can be said that the min-sum algorithm is less computationally complex than BCJR, but more iterations are needed to achieve an acceptable communications performance.

Implementation concerns

LDPC decoding can be parallelized to a very high degree. The belief propagation algorithm allows processing of all check node updates or variable node updates for a complete code word in parallel, meaning we could perform one such half-iteration in just one clock cycle. Unfortunately this requires a very sophisticated communication network, that can route all messages to the respective target nodes at once. Since every LDPC code has a different connectivity, such a design would be limited to one unique application. In most practical cases, however, it is necessary to support multiple block sizes and code rates, making a fully connected decoder prohibitively complex. We are forced to sacrifice throughput for flexibility, but it should be kept in mind that for specialised high-throughput applications the possibility of fully parallel decoding exists for LDPC codes.
Since this approach is not relevant for most designs, partially parallel methods are preferred in combination with specially designed structured codes such as quasi-cyclic (QC) codes [Foss00]. The parity-check matrix of these codes can be divided into $n_b \times n_m$ submatrices of size $z \times z$. These submatrices are either all-zero or permutations of the identity matrix, which means it is not hard to design an interconnect for these very regular structures that can be configured to accommodate a wider selection of different parity-check matrices to enable usage of various block sizes and code rates. The parallelism level of decoding QC-LDPC codes is not limited, to z though. It is possible to process several submatrix rows in parallel, but if they are not independent we suffer a loss in decoding performance as we 'dilute' the layered decoding schedule and move in the direction of TPMP where all submatrix rows would be processed in parallel, regardless of dependencies. This tradeoff can be worked around somewhat by trying to optimize the processing order of submatrices ([Che04]).
Similar to turbo decoding, LDPC decoders profit from the application of early stopping criteria to limit the number of iterations to a minimum [Kie05].

3 Merging Decoding Schemes

In the last chapter we had a look at three algorithms that are commonly used to implement channel decoding. The typical approach to a decoder implementation that is to be used in a communications SoC is to select one algorithm and design an optimized IP block for that algorithm. If the system specification contains more than one channel coding scheme, a dedicated decoder is used for each of them. Now the immediately obvious question arises: Can't we combine these decoders somehow to simplify the system and save chip area and/or power? Some research has been done on this topic in recent years with notable results such as [All08], [Sun08], [Nae08], [Gen10]. In the following sections, we will take a look at combining the functionality of several IP cores in a more general way.

We want to analyze the merging of the hardware implementation of two (or more) algorithms, say we have an algorithm a that is mapped on a hardware block A and an algorithm b mapped on a block B. We now want to create a hardware block C, that enables the processing of a as well as b, with the side constraints of minimizing area and power consumption of C and maximizing the algorithm performance of a and b. We will make the following assumptions throughout this chapter: We focus here on algorithms of a certain minimum complexity which would usually be implemented as stand-alone IP block or coprocessor within an MPSoC environment. We further assume that the algorithm implementation can be represented by a basic architectural model, that is on its highest level made up of memory blocks for value storage and logic cores for data processing. Such a core should contain all functionality for one algorithm and consist of lower-level functional units internally.

To solve this problem, there are two things we need: (1) We need to analyze the target algorithms and identify the potential for merging and (2) we need merging methods we can apply to the target algorithms to exploit the potential identified in the analysis. In this chapter, we are going to present merging methods categorized into different hierarchical levels: core level, functional unit (FU) level, intra-FU level and we are also going to present approaches for analyzing algorithms. But before we start to discuss ways to merge logic blocks, we will first take a moment to see how we can measure the result of such a merging process in regards to its